



Solitaire

Adding Help to a Game

Constructing a 3D Game of Skill

Recording Game State

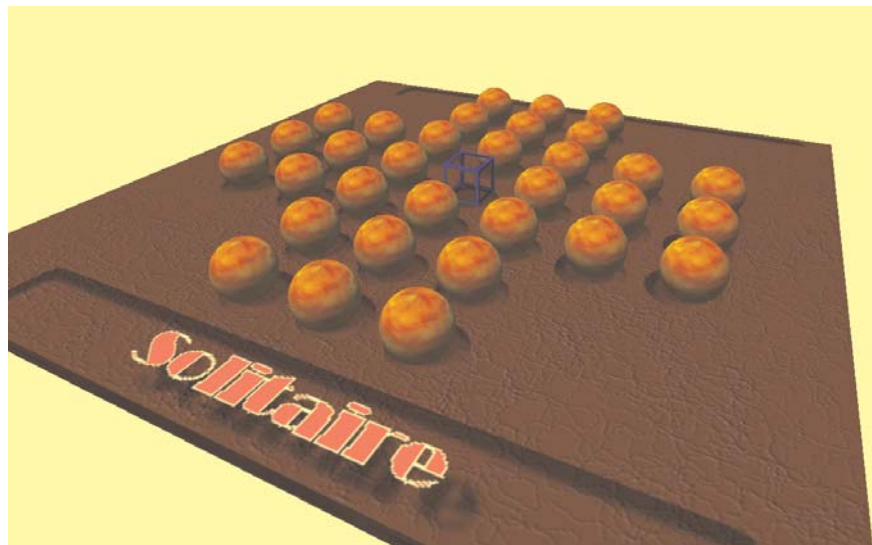
Solitaire - The Board Game

Introduction

In this chapter we are going to develop a 3D version of the board game Solitaire (see FIG-38.1) - not to be confused with the card game of the same name.

FIG-38.1

The Game of Solitaire



The Equipment

The game consists of 32 marbles and a board with 33 indentations, or pits. The pits form the shape of a plus sign (+). One marble is placed in all but one pit, the central pit remaining empty.

The Aim

The aim of the game is to remove all but one marble from the board.

The Rules

Any marble can be moved by jumping over one other marble into an empty pit. Jumps can be horizontal or vertical but not diagonal. Only a single marble can be jumped on each move. The marble which is jumped over is removed from the board.

Creating a Computer Version of the Game

User Controls

Each turn involves the player first selecting a marble and then an empty pit into which the marble is to be moved.

To achieve this, the player will move a selector cube about the board using the arrow keys. Once over a required pit, the Enter key is pressed to indicate a selection.

Pressing U will allow the player to deselect a marble.

Pressing N will initiate a new game.

Pressing F will terminate the current game.

Pressing F1 will display the help screen. Within the help screen, pressing R will display the rules, and K the keys which may be used in the game.

Game Responses

The game will respond to player actions in the following ways:

- When a marble is selected, it will change colour.
- When an empty pit is selected as the second part of a move, the selected marble will jump to the new position and return to its original colour. The jumped marble will be removed.
- When U is pressed immediately after a marble has been selected, the selected marble will return to its original colour and the player will have to select a new marble.
- If an empty pit is selected when marble selection is required, an error message will be displayed, and the player must select a pit containing a marble.
- If an occupied pit is selected when an empty pit is required, an error message is displayed, and a new destination pit must be selected.
- If the marble and destination pit selected do not generate a valid move, an error message is displayed, and a new destination pit must be selected.
- If N is pressed, a new game is started. No prompt is given.
- If F is pressed, the program will terminate after displaying a shut down message.
- If F1 is pressed, the introductory page of the Help section will be displayed. A second press of the same key will cause the help screen to disappear.
- If R is pressed while a help page is visible, the help page containing the rules will be displayed.
- If K is pressed while a help page is visible, the help page showing the keys which can be used in the game will be displayed.

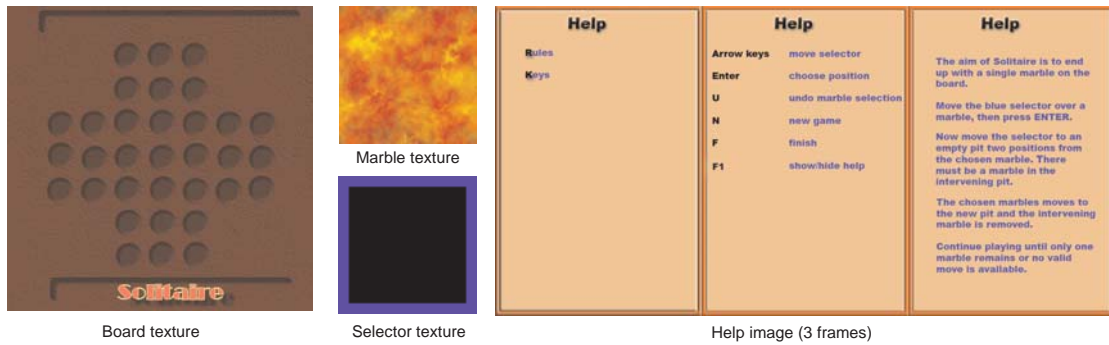
Screen Layout

The screen layout is shown in FIG-38.1.

Media Used

The images used to texture the board, marbles and selector cube are shown in FIG-38.2. Notice that even the help text is held as a single image with three frames.

FIG-38.2 The Images Used in the Program



Data Structures

Although we can use 3D objects to represent the board visually, we need a second way of representing the board as data. This will make the coding easier when trying to decide on valid moves.

If we had a square board, the obvious approach would be to use a two-dimensional array with one cell representing one pit on the board. However, with a little thought, we can still do that. In the centre the board is 7 pits wide and 7 pits deep, so a 7 by 7 integer array could be used to represent the board:

```
DIM board (7,7)
```

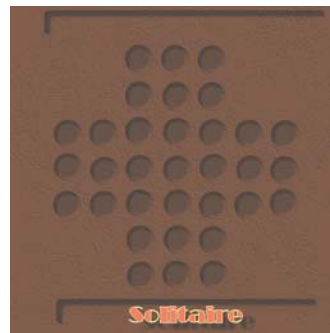
Now we'll fill the array with a zero in any cell that corresponds to a pit and with -1 where a cell corresponds to a flat part of the board. FIG-38.3 shows the board and the corresponding array contents.

FIG-38.3

The Board and the Array used to Represent it

-1	-1	0	0	0	-1	-1
-1	-1	0	0	0	-1	-1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
-1	-1	0	0	0	-1	-1
-1	-1	0	0	0	-1	-1

board Array



Actual Board Layout

Rather than just store zeros, it will suit us better to store the IDs of each marble in the cells of the array. So, if we assume the marbles have object IDs starting at 11, the board array would start off as shown in FIG-38.4.

FIG-38.4

The *board* array with Marble IDs

-1	-1	11	12	13	-1	-1
-1	-1	14	15	16	-1	-1
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	32	33	34	35	36	37
-1	-1	38	39	40	-1	-1
-1	-1	41	42	43	-1	-1

board Array

It would be useful to employ a second 7 by 7 array in which to store the world coordinates of each pit. Although we could work the positions out mathematically (since the pits are evenly spaced) it will probably create less complex code if we just store the coordinates of each position in another array. Since the board is flat, it won't be necessary to store the y value of any position (since these will all be identical), just the x and z values.

When we create a 3D object, its centre is at (0,0,0), so, assuming we don't move the board, the centre pit will have x, z values of 0,0.

Since we need to store two values in every cell of the array (the x and z ordinates), we'll need to create a record structure for this

```

TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

```

before creating our array:

```

DIM boardCoords(7,7) AS Coords

```

Now, assuming the centre of each pit is 3 world units in both directions from any neighbour, the new array would contain the values shown in FIG-38.5.

FIG-38.5

The *boardCoords* Array

-1	-1	(-3,9)	(0,9)	(3,9)	-1	-1
-1	-1	(-3,6)	(0,6)	(3,6)	-1	-1
(-9,3)	(-6,3)	(-3,3)	(0,3)	(3,3)	(6,3)	(9,3)
(-9,0)	(-6,0)	(-3,0)	(0,0)	(3,0)	(6,0)	(9,0)
(-9,-3)	(-6,-3)	(-3,-3)	(0,-3)	(3,-3)	(6,-3)	(9,-3)
-1	-1	(-3,-6)	(0,-6)	(3,-6)	-1	-1
-1	-1	(-3,-9)	(0,-9)	(3,-9)	-1	-1

boardCoords Array

In practice, we'll probably store coordinates in the invalid positions of the array, since it is easier to write the code that way.

We'll also need to record the position of the selector cube. But rather than store this in world coordinates, we'll just store which row and column of the board it's currently placed at. For this we'll need another record structure:

```

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

```

then we can declare a variable for the selector's position:

```

selectorposition AS BoardPosition

```

On each move the player chooses a marble and a position to which the marble is to

be moved. We'll need to record these two positions as well:

```
GLOBAL DIM moves(2) AS BoardPosition
```

The final important variable that we will find ourselves needing is a game-state variable. There are three main stages during game play. These are:

- About to choose a marble
- About to choose a move-to space
- Looking at the help text

We need to be aware of which state the game is in, because this dictates what options are currently available to the player. When the player is about to choose a marble, it would be invalid to select an empty pit; however, choosing a pit containing a marble would be invalid when the player is about to choose the move-to position. Trying to play while viewing the Help is also disallowed.

To record what state the game is in at any moment, we have another variable which can be assigned only the values 1, 2 or 3:

```
gamestate '1-choose marble; 2-choose move-to; 3-showing help
```

Lastly, we need to record how many marbles are currently on the board; when this reduces to one, the game is over.

All the variables above will be made global so they can be accessed throughout the program:

```
REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)           `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords `Marble coordinates
GLOBAL selectorposition AS BoardPosition `Position of selector
GLOBAL DIM move(2) AS BoardPosition `marble and move-to
                                   `positions
GLOBAL gamestate                `Game state
                                   `1-choose marble;
                                   `2-choose space;
                                   `3-showing help
GLOBAL marblesRemaining         `marbles on board
```

The images and 3D objects also need to be assigned ID values and it will make the code clearer if these are given names in #CONSTANT statements.

```
REM *****
REM ***          Constants          ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj          1
#CONSTANT selectorobj      2
#CONSTANT firstmarbleobj   3
```

```

#CONSTANT lastmarbleobj      34
REM *** Image constants ***
#CONSTANT boardimg          1
#CONSTANT marbleimg         2
#CONSTANT selectorimg       3
#CONSTANT helpimage         4
REM *** Sprite constants ***
#CONSTANT helpsprite        1

```

Now it's time to have a look at the logic of the game. As usual, for a larger project, the main section should consist mainly of function calls so we can get an overview of what's going on in just a few lines. In English the whole logic could be described as follows:

```

Initialise screen
Set up game
REPEAT
    Get player move
UNTIL game over

```

In the main, this translates into straight function calls but we'll have to know if the player has chosen to quit the game, so the function which gets the player's move will have to return a flag to indicate if the game should be ended. This means that the main section should be written as:

```

ScreenSetUp()
GameSetUp()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
END

```

Activity 38.1

Create a new project (*Solitaire.dbpro*) and copy the code for the constants and data into the source file.

Code the main section as given above and add test stubs for each of the functions called. The test stub for *GetPlayerMove()* should be:

```

FUNCTION GetPlayersMove()
    PRINT "GetPlayersMove() executed"
ENDFUNCTION 1

```

Run the program and make sure it executes.

To stop the program finishing before you get time to see all the messages displayed, add a `WAIT KEY` statement before the `END` command in the program's main section.

Make sure all of the media files have been copied to the folder.

Now we'll begin to build up the project by working our way through each routine needed.

Adding *ScreenSetUp()*

This routine sets the screen resolution and backdrop as well as positioning the cameras, so its code is quite straight forward:

```

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

```

Activity 38.2

Replace the test stub for *ScreenSetUp()* with the actual code as given above.

Once a 3D object (or sprite) appears on the screen, or we start using a camera, the screen refresh method employed means that the output from any PRINT statement vanishes almost as soon as it appears. Because of this, we can no longer see the messages confirming that routines have been executed.

You might feel that we no longer need to see these messages now that we're up and running but, unfortunately, you're probably wrong! Remember every program is just waiting its opportunity to catch us out - so don't take any chances!

There are various ways we might overcome the problem of the disappearing messages, but in this case we're going to create a replacement for the PRINT statement that displays a string at any point on the screen for a given number of seconds. The new function, *DisplayMessage()* is coded below:

```

FUNCTION DisplayMessage(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y
    PRINT message$
  ENDWHILE
ENDFUNCTION

```

Activity 38.3

Add *DisplayMessage()* to the end of your program and replace all the PRINT statements in the test stubs with calls to this function.

Adding *GameSetUp()*

This routine is a bit more complex since we need to set up the board, add the marbles, position the selector and set up the help feature. Since each of these might take up several lines of code, it might again be best if we create new routines to handle each stage. This means that the *GameSetUp()* function itself just consists of calls to other functions:

```

FUNCTION GameSetUp()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  HelpSetup()
  gamestate = 1
ENDFUNCTION

```

Activity 38.4

Add the code for *GameSetUp()* to your program and add test stubs for the routines that it calls.

Adding *CreateBoard()*

This routine is responsible for setting up the board. This means, not only does it create a thin 3D box to represent the board, texturing it with the board image shown in FIG-38.2, but it also fills the *board* and *boardCoords* arrays with relevant data.

While the visual aspects of creating the board requires only a few lines, initialising the arrays is much more complex, so again we'll make another routine to do that part of the job (*CreateInternalBoard()*). The code for *CreateBoard()* is:

```
FUNCTION CreateBoard()  
  REM *** Set up board arrays ***  
  CreateInternalBoard()  
  REM *** Create and texture board ***  
  MAKE OBJECT BOX boardobj,27,27,0.5  
  LOAD IMAGE "board.bmp",boardimg  
  TEXTURE OBJECT boardobj,boardimg  
  REM *** Turn and position board ***  
  ROTATE OBJECT boardobj,90,0,0  
  POSITION OBJECT boardobj,-0.2,-1.2,0  
ENDFUNCTION
```

Activity 38.5

Update your program with the code above.

Create a test stub for *CreateInternalBoard()*.

Adding *CreateInternalBoard()*

Although this is a rather long routine, it doesn't do anything too complicated; just fills the *board* and *boardContents* arrays with the values shown earlier. The code for the routine is:

```
FUNCTION CreateInternalBoard()  
  REM *** Initialise the board array ***  
  #CONSTANT EMPTYFIT 0  
  #CONSTANT NOTALLOWED -1  
  REM *** Assume every position empty ***  
  FOR row = 1 TO 7  
    FOR col = 1 TO 7  
      board(row,col) = EMPTYFIT  
    NEXT col  
  NEXT row  
  REM *** then mark the invalid positions ***  
  board(1,1) = NOTALLOWED  
  board(1,2) = NOTALLOWED  
  board(2,1) = NOTALLOWED  
  board(2,2) = NOTALLOWED  
  board(1,6) = NOTALLOWED  
  board(1,7) = NOTALLOWED  
  board(2,6) = NOTALLOWED  
  board(2,7) = NOTALLOWED  
  board(6,1) = NOTALLOWED  
  board(6,2) = NOTALLOWED
```

```

board(7,1) = NOTALLOWED
board(7,2) = NOTALLOWED
board(6,6) = NOTALLOWED
board(6,7) = NOTALLOWED
board(7,6) = NOTALLOWED
board(7,7) = NOTALLOWED

REM *** Initialise the boardContents array ***
REM *** with the pit coordinates ***
x = -9
z = 9
FOR row = 1 TO 7
  FOR col = 1 TO 7
    BoardCoords(row,col).x = x
    BoardCoords(row,col).z = z
    x = x + 3
  NEXT col
  z = z - 3
  x = -9
NEXT row
ENDFUNCTION

```

Activity 38.6

Add the *CreateInternalBoard()* function to your program.

Although it won't make any visible difference to your program, execute the code just to make sure there are no syntax errors or other problems.

Adding *CreateMarbles()*

Now we need to create the 3D marble objects. These will be placed just above the pits on the board, except for the central pit which remains empty.

The logic for this routine can be described in structured English as:

```

Load marble texture
Set object number to firstmarbleobj
FOR each row on the board
  FOR each column on the board
    IF the pit is empty AND NOT the central pit THEN
      Create sphere (2 units diameter)
      Texture sphere
      Position sphere at the coordinates given in boardCoords(row,col)
      Store the sphere's ID value in board(row,col)
      Increment objno
    ENDIF
  ENDFOR
ENDFOR

```

Activity 38.7

Use the logic given above to produce the code for the *CreateMarbles()* function and test your updated program.

Adding *CreateSelector()*

A blue outline cube is used to select which marble is to be moved and to which pit it is to move. The *CreateSelector()* creates the cube placing it initially in the central pit. The logic for the routine is:

Load selector texture
 Create selector cube object (1.5 units)
 Texture cube
 Make black areas of cube invisible
 Switch off culling so back lines are drawn
 Position the cube at central pit
 Set *selectorposition* to row 4, column 4

Activity 38.8

Use the logic given to create the final version of the *CreateSelector()* routine and add it to your program.

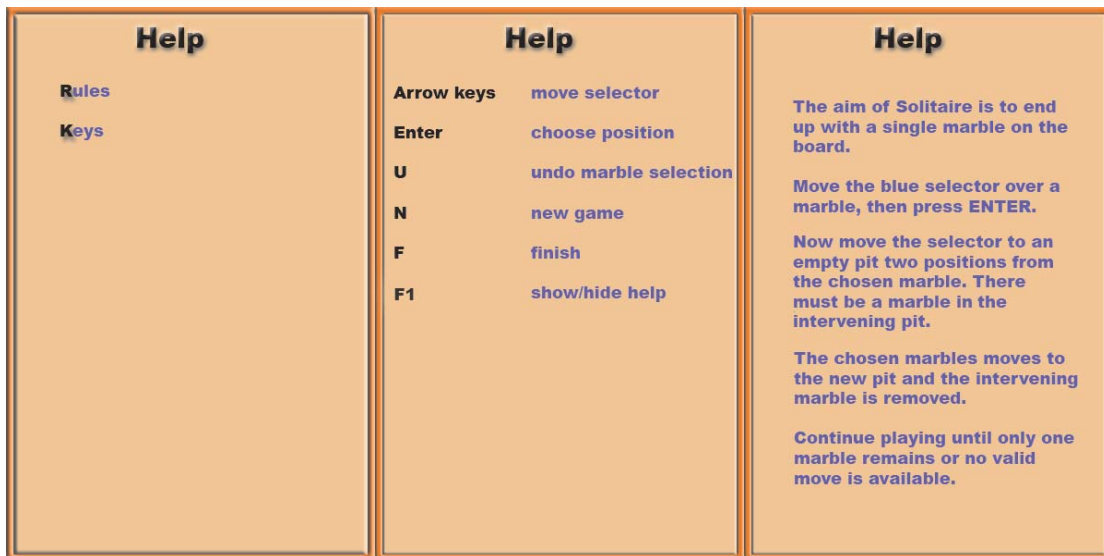
Check that the selector appears at the correct position on the board.

Adding *HelpSetUp()*

The player will be able to get help on the game rules and on which keys can be used by pressing the F1 key (the traditional help key for Microsoft Windows programs).

The help itself will be created as an animated sprite containing three frames; one for each page of the help (see FIG-38.6).

FIG-38.6 The Help Image



This routine will create the required sprite, but then hide the sprite. The sprite will appear whenever the F1 key is pressed. If the user presses the F1 key a second time, the help sprite will disappear.

If you want to create your own image using a paint package, remember not to use any black since this will automatically become transparent when the sprite is displayed.

The code for this routine is given below:

```
FUNCTION HelpSetup()  
  REM *** Create the sprite required by help ***  
  CREATE ANIMATED SPRITE helpsprite,"rules.bmp",3,1,helpimg  
  REM *** Position sprite ***
```

```

        SPRITE helpsprite,10,10,helpimg
        REM *** Set sprite to show first page ***
        SET SPRITE FRAME helpsprite,1
        REM *** Hide sprite ***
        HIDE SPRITE helpsprite
    ENDFUNCTION

```

Of course, we'll need another routine later to make the *Help* sprite appear when required.

Activity 38.9

Add the *HelpSetUp()* function to your program.

Check that the program still compiles correctly.

Adding *GetPlayerMove()*

The *GetPlayerMove()* is the core routine of the game, so it will probably come as no surprise that it is the most complex. However, it really only involves reading the keyboard, recognising which key has been pressed and carrying out the required action if it is valid. Invalid actions will produce error messages.

Unlike all the other functions, this one returns a 1 if the user has chosen the quit option and zero for all other options.

The logic for this routine is:

```

Set quit to zero
Read a key
IF
    arrow key pressed:
        Move selector
    Enter key pressed:
        IF
            gamestate = 1:
                Select a marble
                IF okay THEN
                    Set gamestate to 2
                ENDIF
            gamestate = 2:
                Select a pit
                IF okay THEN
                    Move marble
                    Set gamestate to 1
                ENDIF
        ENDIF
    U pressed:
        IF state = 2 THEN
            Undo marble selection
            Set gamestate to 1
        ENDIF
    N pressed:
        Restart game
    F1 pressed:
        IF gamestate < 10 THEN
            Show first page of help
            Add 10 to gamestate
        ELSE
            Hide help
            Subtract 10 from gamestate
        ENDIF
    K pressed:
        IF gamestate > 10 THEN

```

```

        Show keys help page
    ENDIF
R pressed:
    IF gamestate > 10 THEN
        Show rules help page
    ENDIF
F pressed:
    Set quit to 1
ENDIF
Wait for all keys to be released

```

The code for this routine is:

```

FUNCTION GetPlayersMove()
    REM *** Assume player will not quit this turn ***
    quit = 0
    REM *** Read a key ***
    REPEAT
        code = SCANCODE()
    UNTIL code <> 0
    SELECT code
        CASE 200,208,205,203      `Arrow keys
            MoveSelector(code)
        ENDCASE
        CASE 28                  `Enter key
            REM *** IF at select marble stage, select marble ***
            IF gamestate = 1
                REM *** IF marble selected, gamestate = 2, ***
                IF SelectMarble()
                    gamestate = 2
                ENDIF
            ELSE
                REM *** IF in select empty pit stage,select pit
                IF gamestate = 2
                    REM *** IF pit okay... ***
                    IF SelectPit()
                        REM *** Move marble and return to state 1
                        MoveMarble()
                        gamestate = 1
                    ENDIF
                ENDIF
            ENDIF
        ENDCASE
        CASE 22                  `U - undo selection
            REM *** IF marble selected THEN ***
            IF gamestate = 2
                REM *** Return marble to normal colour ***
                objno = board(move(1).row,move(1).col)
                TEXTURE OBJECT objno,marbleimg
                REM *** return to gamestate 1 ***
                gamestate = 1
            ENDIF
        ENDCASE
        CASE 49                  `N - new game
            ScreenSetUp()
            GameSetUp()
        ENDCASE
        CASE 59                  `F1 - Help
            IF gamestate < 10
                SelectHelpPage(1)
                gamestate = gamestate + 10
            ELSE
                SelectHelpPage(0)
                gamestate = gamestate - 10
            ENDIF
        ENDCASE
        CASE 37                  `K - Help page 2
            IF gamestate > 10

```

```

        SelectHelpPage(2)
    ENDIF
ENDCASE
CASE 19          `R - Help page 3
    IF gamestate > 10
        SelectHelpPage(3)
    ENDIF
ENDCASE
CASE 33          `F - Quit game
    quit = 1
ENDCASE
ENDSELECT
REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
    ENDWHILE
ENDFUNCTION quit

```

Activity 38.10

Add the *GetPlayerMove()* function to your program.

Add test stubs for any routines called by *GetPlayerMove()* that are yet to be written.

Test the program.

Adding *MoveSelector()*

The *MoveSelector()* function takes a parameter. This is the arrow key code and is used to decide in which direction the selector is to be moved.

The function's logic is:

```

Set row to copy of selectorposition.row
Set col to copy of selectorposition.col
IF
    left arrow AND not at left hand column :
        Decrement col
    right arrow AND not at right-hand column:
        Increment col
    down arrow AND not at bottom row:
        Decrement row
    up arrow AND not at top row:
        Increment row
ENDIF
IF board(row,col) is a valid position THEN
    Set selectorposition.row to copy of row
    Set selectorposition.col to copy of col
    Move selector to new position
ENDIF

```

and this is coded as:

```

FUNCTION MoveSelector(code)
    REM *** Move selector in response to arrow key ***
    row = selectorposition.row
    col = selectorposition.col
    IF code = 203 AND col>1
        DEC col
    ELSE IF code = 205 AND col < 7
        INC col
    ELSE IF code = 200 AND row > 1
        DEC row
    ELSE IF code = 208 AND row < 7

```

```

        INC row
    ENDIF
ENDIF
ENDIF
ENDIF
REM *** IF a valid position, move selector ***
IF board(row,col) <> -1
    selectorposition.row = row
    selectorposition.col = col
    POSITION OBJECT selectorobj,
    BoardCoords(row,col).x,0,BoardCoords(row,col).z
ENDIF
ENDFUNCTION

```

Activity 38.11

Add the new routine to your program and check that the selector moves correctly.

Adding *SelectMarble()*

In the first phase of a move (when *gamestate* = 1), the player has to select a marble. This involves the player moving the selector over the required pit position and then pressing *Enter*. At this point, control jumps to the *SelectMarble()* routine which checks that the selected pit does, in fact, contain a marble (if not, an error message is displayed) and changes the colour of the marble to indicate its selection. The routine returns 1 when a marble has been selected and zero when no marble was selected.

The logic for this routine is:

```

IF selector not over marble THEN
    Display an error message
    Return 0 from function
ENDIF
Change colour of selected marble
Record position of marble in move(1)
Return 1 from function

```

and is coded as:

```

FUNCTION SelectMarble()
    REM *** IF selector not over marble THEN ***
    IF board(selectorposition.row,selectorposition.col) < 1
        REM *** Display message; return 0 ***
        DisplayMessage(100,100,"Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
    REM *** Change marble colour ***
    COLOR OBJECT
    board(selectorposition.row,selectorposition.col),
    RGB(255,255,0)
    REM *** record position of marble ***
    move(1).row = selectorposition.row
    move(1).col = selectorposition.col
ENDFUNCTION 1

```

Activity 38.12

Add *SelectMarble()* to your program and check that it functions correctly.

Adding *SelectPit()*

The second part of a move involves choosing an empty pit into which the selected marble is to be moved.

As well as the obvious check that the selected pit must be empty, it must also be exactly two pits from the selected marble in either a horizontal or vertical direction. But even that's not all the checking required: moving the marble to the selected pit must involve "jumping over" one other marble (the one that is to be removed).

The logic for the routine is:

```
IF selected pit not empty THEN
  Display "Choose empty pit"
  Return 0 from function
ENDIF
Record selector position in move(2)
IF not a valid move THEN
  Display "Invalid move"
  Return 0 from function
ENDIF
```

Checking for a valid move in the final IF mentioned above will be quite complex, so we'll use another function, *isValidMove()* to carry out the task. This function will return 1 when the move is valid and zero when it's invalid.

Activity 38.13

Using the logic given, add the *SelectPit()* function to your program.

Add a test stub for *isValidMove()* which returns the value 1.

Adding *IsValidMove()*

There are three checks needed in this function:

- That the selected marble is being moved 2 places horizontally
- That the selected marble is being moved 2 places vertically
- That the move involves "jumping" another marble

One of the first two tests must be true and the third test must also be true in order for the move to be a valid one.

Activity 38.14

Write the complete version of the *IsValidMove()* function and test it within your program.

Adding *MoveMarble()*

Once a valid move has been entered, the selected marble must be moved to its new position. On the screen, this not only involves moving the selected 3D object, but also changing it back to its original texture and removing the "jumped" marble. But we also need to make the equivalent changes to the *board* array to reflect the new situation. Also, since a marble is removed from the board, the *marblesremaining*

variable must be decremented.

The logic required by the routine is:

```
Move marble in board array
Move marble on screen
Return marble to original texture
Calculate position of jumped marble
Remove jumped marble from screen
Remove jumped marble from board array
Decrement marblesremaining
```

Activity 38.15

Code the *MoveMarble()* function and add it to your program.

Adding *SelectHelpPage()*

The final routine to be created is the *SelectHelpPage()* function. This function takes a parameter (*pageno*) specifying the page number to be displayed. Since the help pages are stored as frames in an animated sprite, the *pageno* parameter represents the frame to be displayed. If *pageno* is outside the range 0 to 3 then the function exits; if the *pageno* is zero, the help sprite is removed from the screen.

The function requires the following logic:

```
IF invalid value for pageno THEN
    EXIT
ENDIF
IF pageno > 0 THEN
    Set sprite frame to pageno
ELSE
    Hide help sprite
ENDIF
```

Activity 38.16

Code the *SelectHelpPage()* from the logic given and test your completed program.

Using the Mouse

Introduction

An alternative way of selecting a marble and the pit it is to be moved to would be to use the mouse. As we saw in the last chapter, the mouse can be used in conjunction with the PICK OBJECT statement to select 3D objects on the screen. However, although this would work fine for selecting the marble, picking the destination pit for the marble would prove more difficult, since there is no object within an empty pit that can be selected with a mouse click.

One way to overcome this problem is to place an invisible object at every valid pit position. The PICK OBJECT statement can be used to select invisible objects. This means that pits which show a marble will actually contain two objects: a marble and an invisible object; pits without a marble will contain only an invisible object.

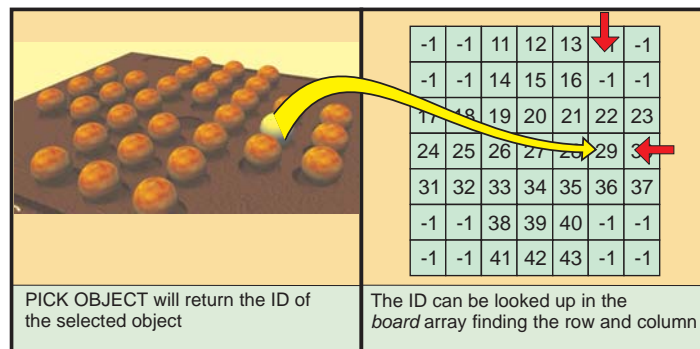
To make a move the player will first click on a pit containing a marble and the mouse

position will be compared with the range of marble IDs to discover which one has been selected. When the destination is being selected as the second part of a move, the mouse position will be compared with the set of invisible objects.

Although this will give us the ID values of the marble and invisible objects, we still have to translate this into row and column values so that the *board* array can be updated and the *move* array can be filled. We can find the selected marble's row and column position by searching the *board* array for a match with the ID returned by the PICK OBJECT statement (see FIG-38.7).

FIG-38.7

Finding the ID of the Selected Marble



If we set up an array containing the IDs of the invisible objects, we can use the same technique to find the row/column position of the empty pit selected as the second part of a move.

Updating the Program

Activity 38.17

Create a new project (*Solitaire2.dbpro*) and copy the source code for the first version of the game into the new source file.

Copy all of the media files used into the new folder.

With a copy of the original program, it won't matter if things go wrong - we can always go back.

We need to start by removing all references to the selector object since it won't be needed in the new version of the game.

Activity 38.18

Remove all lines containing references to *selectoring*, *selectorobj*, and *selectorposition* from the program.

The next step is to add our invisible objects to each pit. We can't hide the objects since PICK OBJECT doesn't detect hidden objects but we can texture the objects with a black image and then use SET OBJECT TRANSPARENCY to make the objects disappear.

To do this we'll create some new constants:

```
#CONSTANT hiddenimg      3
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj  67
```

and an array in which to store the IDs:

```
GLOBAL DIM PitObjects(7,7)           `IDs of the hidden objects
```

The routine to create the hidden objects would then be coded as:

```
FUNCTION CreateHiddenPitObjects()  
  REM *** Set first hidden object ID ***  
  pitobjno = firsthiddenobj  
  REM *** Load image used to texture objects ***  
  LOAD IMAGE "black.bmp",5  
  REM *** FOR each position on board ***  
  FOR row = 1 TO 7  
    FOR col = 1 TO 7  
      REM *** IF its a valid position ***  
      IF board(row,col) > -1  
        REM *** Record hidden object ID for that pit  
        PitObjects(row,col) = pitobjno  
        REM *** Make, texture and hide object ***  
        MAKE OBJECT SPHERE pitobjno,2  
        TEXTURE OBJECT pitobjno,5  
        POSITION OBJECT pitobjno,boardCoords(row,col).x,  
        0,boardCoords(row,col).z  
        SET OBJECT TRANSPARENCY pitobjno,1  
        REM *** Increment object ID  
        INC pitobjno  
      ENDIF  
    NEXT col  
  NEXT row  
ENDFUNCTION
```

This routine should be called from *GameSetUp()*.

Activity 38.19

Add the constants and global variables mentioned above to your program.

Add the routine and place a call to it in *GameSetUp()*.

Copy the file *black.bmp* to your folder.

Now we have to make some changes to the *GetPlayersMove()* function. The original version of this function reacts to key presses, but in the update we need to allow for both mouse and key presses to be detected. The routine starts by waiting for a key press using the lines

```
REPEAT  
  code = SCANCODE()  
UNTIL code <> 0
```

but in the new version, we need to wait for a key press or a mouse click and this is achieved by the code:

```
REPEAT  
  mouse = MOUSECLICK()  
  code = SCANCODE()  
UNTIL code <> 0 OR mouse <> 0
```

The SELECT statement that follows was based on the scan code of the key pressed. Since we no longer use a selector, detecting the arrow keys is no longer required, so the lines

```

CASE 200,208,205,203    `Arrow keys
ENDCASE

```

can be removed from this new version of the program.

The next CASE statement within the SELECT is CASE 28 which was used to detect the *Enter* key. But now the mouse button needs to activate this option. To keep our changes to a minimum we can set *code* to 28 when the mouse is clicked:

```

IF mouse <> 0
    code = 28
ENDIF

```

These lines need to be placed just before the SELECT structure.

At the end of the SELECT structure we waited for the button pressed by the player to be released using the code

```

REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
ENDWHILE

```

but now we also need to wait for the mouse button to be released:

```

REM *** Wait till all keys and mouse buttons released ***
WHILE SCANCODE() <> 0 OR MOUSECLICK() <> 0
ENDWHILE

```

Activity 38.20

Make the changes described to the *PlayerMove()* routine.

The *SelectMarble()* routine needs to be changed. This version needs to detect the mouse pointer position and select the marble over which it has been placed. The logic for the updated routine is:

```

Find mouse coordinates
Get ID of marble at this screen position
IF no ID detected THEN
    Display "Choose a marble"
    Return 0 from the routine
ENDIF
Find position of ID in board array and record row and column
Change colour of marble selected
Record row and column in moves()
Return 1 from routine

```

This is coded as:

```

FUNCTION SelectMarble()
    REM *** Check if position chosen contains marble ***
    x = MOUSEX()
    y = MOUSEY()
    objno = PICK OBJECT(x,y,firstmarbleobj,lastmarbleobj)
    IF objno = 0
        DisplayMessage(100,100,"Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
    REM *** Find marble's ID in board array ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7

```

```

        IF board(row,col) = objno
            prow = row
            pcol = col
        ENDIF
    NEXT col
NEXT row
REM *** Change marble colour ***
COLOR OBJECT objno,RGB(255,255,0)
REM *** record position of marble
move(1).row = prow
move(1).col = pcol
ENDFUNCTION 1 `Valid marble selected

```

The search for the marble ID within the board array is very inefficient coding, but for such a small search area will have no real effect on the speed of the game - and it has the advantage of being quite simple.

Activity 38.21

Update your *SelectMarble()* function to match the code given.

The last routine to be changed is the *SelectPit()* function, which again needs to use the mouse's position to determine which empty pit has been selected.

```

Find mouse coordinates
Get ID of hidden object at this screen position
Get ID of marble at this screen position
IF no hidden ID detected OR a marble ID was detected THEN
    Display "Choose an empty space"
    Return 0 from the routine
ENDIF
Find position of hidden object ID in pitObjects array and record the row and column
Record row and column in moves(2)
Check to see if move is valid
IF not valid THEN
    Display "Invalid move"
    Return 0 from function
ENDIF
Return 1 from routine

```

Activity 38.22

Use the above logic to code the *SelectPit()* function and test your program.

Suggested Enhancements

What we've developed in this chapter is the core of a game, but it still needs all those nice finishing touches that gives a game that professional look.

Some of these would be simple to implement: use a larger font for messages; change the textures used; a introductory splash screen; a finish screen; displaying the number of marbles remaining; displaying the time elapsed; allowing the camera to be moved.

Others would be a bit more complicated: keeping a top 5 best scores (least marbles remaining); allowing the player to backtrack - undoing an unlimited number of moves (you would need a stack structure for this one).

If you have time, see what you can come up with - you'll learn a lot from the exercise.

Solutions

Activity 38.1

```
REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helpimg       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
WAIT KEY `***** Remove later ****
END

`*****
`***** Test stubs*****
`*****
FUNCTION ScreenSetUp()
  PRINT "ScreenSetUp() executed"
ENDFUNCTION

FUNCTION GameSetUp()
  PRINT "GameSetUp() executed"
ENDFUNCTION

FUNCTION GetPlayersMove()
  PRINT "GetPlayerMove() executed"
ENDFUNCTION 1
```

Activity 38.2

```
REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helpimg       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
WAIT KEY `***** Remove later ****
END

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

`*****
`***** Test stubs*****
`*****
FUNCTION GameSetUp()
  PRINT "GameSetUp() executed"
ENDFUNCTION
```

```

FUNCTION GetPlayersMove()
  PRINT "GetPlayerMove() executed"
ENDFUNCTION 1

```

Activity 38.3

```

REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION DisplayMessage
(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000

```

```

WHILE TIMER() - t < millisecs
  SET CURSOR x,y
  PRINT message$
ENDWHILE
ENDFUNCTION

```

```

~*****
~***** Test stubs*****
~*****
FUNCTION GameSetUp()
  DisplayMessage
  (100,100 "GameSetUp() executed",2)
ENDFUNCTION

FUNCTION GetPlayersMove()
  DisplayMessage
  (100,100 "GetPlayerMove() executed",2)
ENDFUNCTION 1

```

The final WAIT KEY statement may also be removed at this stage.

Activity 38.4

```

REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

```

```

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION GameSetUp()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  HelpSetup()
  gamestate = 1
ENDFUNCTION

FUNCTION DisplayMessage
(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y
    PRINT message$
  ENDWHILE
ENDFUNCTION

~*****
~***** Test stubs*****
~*****

FUNCTION GetPlayersMove()
  DisplayMessage
  (100,100 "GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateBoard()
  DisplayMessage(100,100,
  "CreateBoard() executed",2)
ENDFUNCTION

FUNCTION CreateMarbles()
  DisplayMessage(100,100,
  "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
  DisplayMessage(100,100,
  "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION HelpSetup()
  DisplayMessage(100,100,
  "HelpSetup() executed",2)
ENDFUNCTION

```

Activity 38.5

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helpimg 4

```

```

REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION GameSetUp()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  HelpSetup()
  gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
  REM *** Set up board arrays ***
  CreateInternalBoard()
  REM *** Create and texture board ***
  MAKE OBJECT BOX boardobj,27,27,0.5
  LOAD IMAGE "board.bmp",boardimg
  TEXTURE OBJECT boardobj,boardimg
  REM *** Turn and position board ***
  ROTATE OBJECT boardobj,90,0,0
  POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

```

```

FUNCTION DisplayMessage
(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y

```

```

        PRINT message$
    ENDWHILE
ENDFUNCTION

~*****
~***** Test stubs*****
~*****

FUNCTION GetPlayersMove()
    DisplayMessage
    (100,100 "GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateMarbles()
    DisplayMessage(100,100,
    "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
    DisplayMessage(100,100,
    "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION HelpSetup()
    DisplayMessage(100,100,
    "HelpSetup() executed",2)
ENDFUNCTION

FUNCTION CreateInternalBoard()
    DisplayMessage(100,100,
    "CreateInternalBoard() executed",2)
ENDFUNCTION

```

Activity 38.6

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helping 4
REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
    x AS INTEGER
    z AS INTEGER
ENDTYPE

TYPE BoardPosition
    row AS INTEGER
    col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help

```

```

GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetup()
GameSetup()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION ScreenSetup()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280, 1024, 32
    COLOR BACKDROP RGB(255,255,150)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 10,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION GameSetup()
    CreateBoard()
    CreateMarbles()
    CreateSelector()
    HelpSetup()
    gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
    REM *** Set up board arrays ***
    CreateInternalBoard()
    REM *** Create and texture board ***
    MAKE OBJECT BOX boardobj,27,27,0.5
    LOAD IMAGE "board.bmp",boardimg
    TEXTURE OBJECT boardobj,boardimg
    REM *** Turn and position board ***
    ROTATE OBJECT boardobj,90,0,0
    POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
    REM *** Initialise the board array ***
    #CONSTANT EMPTYPIE 0
    #CONSTANT NOTALLOWED -1
    REM *** Assume every position empty ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            board(row,col) = EMPTYPIE
        NEXT col
    NEXT row
    REM *** then mark invalid positions ***
    board(1,1) = NOTALLOWED
    board(1,2) = NOTALLOWED
    board(2,1) = NOTALLOWED
    board(2,2) = NOTALLOWED
    board(1,6) = NOTALLOWED
    board(1,7) = NOTALLOWED
    board(2,6) = NOTALLOWED
    board(2,7) = NOTALLOWED
    board(6,1) = NOTALLOWED
    board(6,2) = NOTALLOWED
    board(7,1) = NOTALLOWED
    board(7,2) = NOTALLOWED
    board(6,6) = NOTALLOWED
    board(6,7) = NOTALLOWED
    board(7,6) = NOTALLOWED
    board(7,7) = NOTALLOWED
    REM *** Initialise boardContents ***
    REM *** with the pit coordinates ***
    x = -9
    z = 9
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            BoardCoords(row,col).x = x
            BoardCoords(row,col).z = z
        NEXT col
    NEXT row

```

```

        x = x + 3
    NEXT col
    z = z - 3
    x = -9
NEXT row
ENDFUNCTION

FUNCTION DisplayMessage
(x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000
    WHILE TIMER() - t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION

~*****
~***** Test stubs*****
~*****

FUNCTION GetPlayersMove()
    DisplayMessage
    (100,100 "GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateMarbles()
    DisplayMessage(100,100,
    "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
    DisplayMessage(100,100,
    "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION HelpSetUp()
    DisplayMessage(100,100,
    "HelpSetUp() executed",2)
ENDFUNCTION

```

Activity 38.7

The code for *CreateMarbles()* is

```

FUNCTION CreateMarbles()
    REM *** Load marble texture ***
    LOAD IMAGE "laval.bmp",marbleimg
    REM *** Create marbles ***
    objno = firstmarbleobj
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = 0 AND
            ((row <> 4) OR (col <> 4))
                MAKE OBJECT SPHERE objno,2,20,20
                TEXTURE OBJECT objno,marbleimg
                POSITION OBJECT
                objno,BoardCoords(row,col).x,0,
                boardCoords(row,col).z
                board(row,col) = objno
                INC objno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

```

Add the code to your program immediately after the *CreateInternalBoard()* function.

If you've entered this separately (rather than modified the test stub), don't forget to remove the test stub version of the routine.

Activity 38.8

The code for *CreateSelector()* is

```

FUNCTION CreateSelector()
    REM *** Load selector texture ***
    LOAD IMAGE "selector.bmp",selectorimg
    REM *** Create selector cube ***
    MAKE OBJECT CUBE selectorobj,1.5
    TEXTURE OBJECT selectorobj, selectorimg
    SET OBJECT TRANSPARENCY selectorobj,1
    SET OBJECT CULL selectorobj,0
    REM *** Position cube ***
    POSITION OBJECT selectorobj,0,0,0
    REM *** Record position of selector ***
    selectorposition.row = 4
    selectorposition.col = 4
ENDFUNCTION

```

The routine should be placed immediately after *CreateMarbles()*.

Activity 38.9

Place *HelpSetUp()* immediately after *CreateSelector()*.

Activity 38.10

Place *GetPlayersMove()* after *HelpSetUp()*.

The following test stubs should also be added:

```

FUNCTION MoveSelector(move)
    DisplayMessage(100,100,
    "MoveSelector() executed",2)
ENDFUNCTION

FUNCTION SelectMarble()
    DisplayMessage(100,100,
    "SelectMarble() executed",2)
ENDFUNCTION 1

FUNCTION SelectPit()
    DisplayMessage(100,100,
    "SelectPit() executed",2)
ENDFUNCTION 1

FUNCTION MoveMarble()
    DisplayMessage(100,100,
    "MoveMarble() executed",2)
ENDFUNCTION

FUNCTION SelectHelpPage(page)
    DisplayMessage(100,100,
    "SelectHelpPage() executed",2)
ENDFUNCTION

```

Activity 38.11

No solution required.

Activity 38.12

No solution required.

Activity 38.13

The code for *SelectPit()* is:

```

FUNCTION SelectPit()
    REM *** IF selected pit not empty, exit

```

```

IF board(selectorposition.row,
selectorposition.col) <> 0
  DisplayMessage(100,100,
  "Choose an empty space",2)
EXITFUNCTION 0
ENDIF
move(2).row = selectorposition.row
move(2).col = selectorposition.col
IF NOT isValidMove()
  DisplayMessage(100,100,"Invalid move",2)
EXITFUNCTION 0
ENDIF
ENDFUNCTION 1

```

The new test stub is:

```

FUNCTION isValidMove()
  DisplayMessage(100,100,
  "isValidMove() executed",2)
ENDFUNCTION 1

```

Activity 38.14

The code for *isValidMove()* is:

```

FUNCTION isValidMove()
  REM *** Invalid if start & finish pits
  REM *** not exactly 2 pits apart ***
  REM *** or no marble in-between ***
  test1 = (ABS(move(1).row-move(2).row)=2)
  AND (move(1).col = move(2).col)
  test2 = (ABS(move(1).col-move(2).col)=2)
  AND (move(1).row = move(2).row)
  midrow = ABS(move(1).row+move(2).row)/2
  midcol = ABS(move(1).col+move(2).col)/2
  test3 = board(midrow,midcol) > 1
  REM *** Return 1 if tests checks okay ***
  IF (test1 OR test2) AND test3
    result = 1
  ELSE `otherwise return zero
    result = 0
  ENDIF
ENDFUNCTION result

```

Activity 38.15

The code for *MoveMarble()* is:

```

FUNCTION MoveMarble()
  REM *** Move marble in board array data ***
  board(moves(2).row,moves(2).col) =
  board(moves(1).row,moves(1).col)
  board(moves(1).row,moves(1).col) = 0
  REM *** Move marble on screen ***
  POSITION OBJECT
  board(moves(2).row,moves(2).col),
  boardCoords(moves(2).row,moves(2).col).x,
  0,boardCoords(moves(2).row,moves(2).col).z
  TEXTURE OBJECT
  board(moves(2).row,moves(2).col),marbleimg
  REM *** Calc position of jumped marble ***
  jumpedrow = (moves(1).row+moves(2).row)/2
  jumpedcol = (moves(1).col+moves(2).col)/2
  REM *** Remove jumped marble from screen ***
  jumpedmarble = board(jumpedrow,jumpedcol)
  HIDE OBJECT jumpedmarble
  DELETE OBJECT jumpedmarble
  REM *** Remove marble from board array ***
  board(jumpedrow,jumpedcol)=0
  REM *** Decrement marblesremaining ***
  DEC marblesremaining
ENDFUNCTION

```

Activity 38.16

The completed program should be coded as:

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helping 4

REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate `Game state
  `1-choose marble;
  `2-choose space;
  `3-showing help

GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION ScreenSetUp()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION GameSetUp()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  HelpSetup()
  gamestate = 1
ENDFUNCTION

```

```

FUNCTION CreateBoard()
  REM *** Set up board arrays ***
  CreateInternalBoard()
  REM *** Create and texture board ***
  MAKE OBJECT BOX boardobj,27,27,0.5
  LOAD IMAGE "board.bmp",boardimg
  TEXTURE OBJECT boardobj,boardimg
  REM *** Turn and position board ***
  ROTATE OBJECT boardobj,90,0,0
  POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
  REM *** Initialise the board array ***
  #CONSTANT EMPTYFIT 0
  #CONSTANT NOTALLOWED -1
  REM *** Assume every position empty ***
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      board(row,col) = EMPTYFIT
    NEXT col
  NEXT row
  REM *** then mark invalid positions ***
  board(1,1) = NOTALLOWED
  board(1,2) = NOTALLOWED
  board(2,1) = NOTALLOWED
  board(2,2) = NOTALLOWED
  board(1,6) = NOTALLOWED
  board(1,7) = NOTALLOWED
  board(2,6) = NOTALLOWED
  board(2,7) = NOTALLOWED
  board(6,1) = NOTALLOWED
  board(6,2) = NOTALLOWED
  board(7,1) = NOTALLOWED
  board(7,2) = NOTALLOWED
  board(6,6) = NOTALLOWED
  board(6,7) = NOTALLOWED
  board(7,6) = NOTALLOWED
  board(7,7) = NOTALLOWED
  REM *** Initialise boardContents ***
  REM *** with the pit coordinates ***
  x = -9
  z = 9
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      BoardCoords(row,col).x = x
      BoardCoords(row,col).z = z
      x = x + 3
    NEXT col
    z = z - 3
    x = -9
  NEXT row
ENDFUNCTION

FUNCTION CreateMarbles()
  REM *** Load marble texture ***
  LOAD IMAGE "laval.bmp",marbleimg
  REM *** Create marbles ***
  objno = firstmarbleobj
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      IF board(row,col) = 0 AND
        ((row <> 4) OR (col <> 4))
        MAKE OBJECT SPHERE objno,2,20,20
        TEXTURE OBJECT objno,marbleimg
        POSITION OBJECT
          objno,BoardCoords(row,col).x,0,
          boardCoords(row,col).z
        board(row,col) = objno
        INC objno
      ENDIF
    NEXT col
  NEXT row
ENDFUNCTION

FUNCTION CreateSelector()
  REM *** Load selector texture ***
  LOAD IMAGE "selector.bmp",selectorimg
  REM *** Create selector cube ***
  MAKE OBJECT CUBE selectorobj,1.5
  TEXTURE OBJECT selectorobj, selectorimg
  SET OBJECT TRANSPARENCY selectorobj,1
  SET OBJECT CULL selectorobj,0
  REM *** Position cube ***
  POSITION OBJECT selectorobj,0,0,0
  REM *** Record position of selector ***
  selectorposition.row = 4
  selectorposition.col = 4
ENDFUNCTION

FUNCTION HelpSetup()
  REM *** Create sprite requ'd by help ***
  CREATE ANIMATED SPRITE helpsprite,
  "rules.bmp",3,1,helpimg
  REM *** Position sprite ***
  SPRITE helpsprite,10,10,helpimg
  REM *** Set sprite to show first page ***
  SET SPRITE FRAME helpsprite,1
  REM *** Hide sprite ***
  HIDE SPRITE helpsprite
ENDFUNCTION

FUNCTION GetPlayersMove()
  REM *** Read a key ***
  REPEAT
    code = SCANCODE()
  UNTIL code <> 0
  quit = 0
  SELECT code
    CASE 200,208,205,203 `Arrow keys
      MoveSelector(code)
    ENDCASE
    CASE 28 `Enter key
      REM *** IF state 1, get marble ***
      IF gamestate = 1
        REM *** Select marble ***
        REM *** IF okay, move to state 2
        IF SelectMarble()
          gamestate = 2
        ENDIF
        REM *** If state 2, select pit
      ELSE IF gamestate = 2
        REM *** IF okay THEN ***
        IF SelectPit()
          REM *** Move marble ***
          MoveMarble()
          REM *** Return to state 1
          gamestate = 1
        ENDIF
      ENDIF
    ENDCASE
    CASE 22 `U - undo selection
      REM *** IF marble selected THEN ***
      IF gamestate = 2
        REM *** return original texture
        objno =
          board(move(1).row,move(1).col)
        TEXTURE OBJECT objno,marbleimg
        REM *** Return to state 1 ***
        gamestate = 1
      ENDIF
    ENDCASE
    CASE 49 `N - new game
      ScreenSetup()
      GameSetup()
    ENDCASE
    CASE 59 `F1 - Help
      REM *** IF help screen not showing
      IF gamestate < 10
        REM *** Show help page 1 ***
        SelectHelpPage(1)
        REM *** In Help state ***

```

```

        gamestate = gamestate + 10
    ELSE
        SelectHelpPage(0)
        gamestate = gamestate - 10
    ENDIF
ENDCASE
CASE 37      `K - Help page 2
    IF gamestate > 10
        SelectHelpPage(2)
    ENDIF
ENDCASE
CASE 19      `R - Help page 3
    IF gamestate > 10
        SelectHelpPage(3)
    ENDIF
ENDCASE
CASE 33      `F - Quit game
    quit = 1
ENDCASE
ENDSELECT
REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
ENDWHILE
ENDFUNCTION quit

FUNCTION MoveSelector(code)
    REM *** Move selector using arrow key ***
    row = selectorposition.row
    col = selectorposition.col
    IF code = 203 AND col>1
        DEC col
    ELSE IF code = 205 AND col < 7
        INC col
    ELSE IF code = 200 AND row>1
        DEC row
    ELSE IF code = 208 AND row<7
        INC row
    ENDIF
ENDIF
ENDIF
ENDIF
REM *** IF a valid, move selector ***
IF board(row,col) <> -1
    selectorposition.row = row
    selectorposition.col = col
    POSITION OBJECT
    selectorobj,BoardCoords(row,col).x,
    0,BoardCoords(row,col).z
ENDIF
ENDFUNCTION

FUNCTION SelectMarble()
    REM *** IF selector not on marble THEN
    IF board(selectorposition.row,
    selectorposition.col) < 1
        REM *** Display message; return 0 ***
        DisplayMessage(100,100,
        "Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
    REM *** Change marble colour ***
    COLOR OBJECT board(selectorposition.row,
    selectorposition.col),RGB(255,255,0)
    REM *** record position of marble
    move(1).row = selectorposition.row
    move(1).col = selectorposition.col
ENDFUNCTION 1

FUNCTION SelectPit()
    REM *** IF selected pit not empty, exit
    IF board(selectorposition.row,
    selectorposition.col) <> 0
        DisplayMessage(100,100,
        "Choose an empty space",2)
        EXITFUNCTION 0
    ENDIF
    move(2).row = selectorposition.row
    move(2).col = selectorposition.col
    IF NOT isValidMove()
        DisplayMessage(100,100,
        "Invalid move",2)
        EXITFUNCTION 0
    ENDIF
ENDFUNCTION 1

FUNCTION IsValidMove()
    REM *** Invalid if start and finish pits
    REM *** not exactly 2 distant ***
    REM *** or no marble between ***
    test1 = (ABS(move(1).row-move(2).row)=2)
    AND (move(1).col = move(2).col)
    test2 = (ABS(move(1).col-move(2).col)=2)
    AND (move(1).row = move(2).row)
    midrow = ABS(move(1).row+move(2).row)/2
    midcol = ABS(move(1).col+move(2).col)/2
    test3 = board(midrow,midcol) > 1
    REM *** Return 1 if tests okay ***
    IF (test1 OR test2) AND test3
        result = 1
    ELSE
        `otherwise return zero
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION MoveMarble()
    REM *** Move marble in board array ***
    board(move(2).row,move(2).col) =
    board(move(1).row,move(1).col)
    board(move(1).row,move(1).col) = 0
    REM *** Move marble on screen ***
    POSITION OBJECT
    board(move(2).row,move(2).col),
    boardCoords(move(2).row,move(2).col).x,
    0,boardCoords(move(2).row,move(2).col).z
    TEXTURE OBJECT
    board(move(2).row,move(2).col),marbleimg
    REM *** Calc jumped marble position ***
    jumpedrow = (move(1).row+move(2).row)/2
    jumpedcol = (move(1).col+move(2).col)/2
    REM *** Remove marble from screen ***
    jumpedmarble = board(jumpedrow,jumpedcol)
    HIDE OBJECT jumpedmarble
    DELETE OBJECT jumpedmarble
    REM *** Remove marble from board ***
    board(jumpedrow,jumpedcol)=0
    REM *** Decrement marblesremaining ***
    DEC marblesremaining
ENDFUNCTION

FUNCTION SelectHelpPage(pageno)
    REM *** IF invalid page selected, exit
    IF pageno < 0 OR pageno > 3
        EXITFUNCTION
    ENDIF
    REM *** IF no zero page no. display page
    IF pageno > 0
        SET SPRITE FRAME helpsprite,pageno
        SHOW SPRITE helpsprite
    ELSE
        REM *** ELSE, hide help ***
        HIDE SPRITE helpsprite
    ENDIF
ENDFUNCTION

FUNCTION DisplayMessage
(x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000
    WHILE TIMER() - t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION

```

Activity 38.17

No solution required.

Activity 38.18

The following lines should be removed from the main section of the program::

```
#CONSTANT selectorobj      2

#CONSTANT selectorimg      3

GLOBAL selectorposition AS BoardPosition
`Position of selector
```

In *GameSetUp()* remove the line

```
CreateSelector()
```

The function *CreateSelector()* should be deleted.

In *GetPlayersMove()*, remove the line

```
MoveSelector(code)
```

from the first CASE structure.

The routine *MoveSelector()* should be removed.

In *SelectMarble()* remove the lines:

```
IF board(selectorposition.row,
selectorposition.col) < 1

COLOR OBJECT board(selectorposition.row,
selectorposition.col),RGB(255,255,0)

move(1).row = selectorposition.row

move(1).col = selectorposition.col
```

In *SelectPit()* remove the lines:

```
IF board(selectorposition.row,
selectorposition.col) <> 0

move(2).row = selectorposition.row

move(2).col = selectorposition.col
```

Activity 38.19

The constants and globals should now be coded as:

```
REM *** Constants ***

REM *** Object Constants ***
#CONSTANT boardobj      1
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj 67

REM *** Image Constants ***
#CONSTANT boarding      1
#CONSTANT marbleimg     2
#CONSTANT hiddenimg     3
#CONSTANT helpimage     4

REM *** Sprite Constants ***
#CONSTANT helpsprite    1
```

```
REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE
```

```
TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE
```

```
REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions on board
GLOBAL DIM BoardCoords(7,7) AS Coords
`Marble world coordinates
GLOBAL DIM moves(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate
`Game state 1-choose marble; 2-choose
space; 3-showing help
GLOBAL marblesremaining
`Marbles left on board
GLOBAL DIM PitObjects(7,7)
`ID of the hidden objects
```

Activity 38.20/22

The complete program code is:

```
REM *****
REM *** Constants ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj 67

REM *** Image constants ***
#CONSTANT boarding      1
#CONSTANT marbleimg     2
#CONSTANT hiddenimg     3
#CONSTANT helpimg       4
REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE
```

```
REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate
GLOBAL gamestate `Game state
`1-choose marble;
`2-choose space;
`3-showing help
```

```

GLOBAL marblesRemaining
`marbles on board
GLOBAL DIM pitObjects(7,7)
`IDs of hidden objects

REM *** Main section ***
ScreenSetUp()
GameSetUp()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesRemaining = 1
REM *** End program ***
END

FUNCTION ScreenSetUp()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,150)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 10,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION GameSetUp()
    CreateBoard()
    CreateMarbles()
    CreateHiddenPitObjects()
    HelpSetup()
    gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
    REM *** Set up board arrays ***
    CreateInternalBoard()
    REM *** Create and texture board ***
    MAKE OBJECT BOX boardobj,27,27,0.5
    LOAD IMAGE "board.bmp",boardimg
    TEXTURE OBJECT boardobj,boardimg
    REM *** Turn and position board ***
    ROTATE OBJECT boardobj,90,0,0
    POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
    REM *** Initialise the board array ***
    #CONSTANT EMPTYFIT 0
    #CONSTANT NOTALLOWED -1
    REM *** Assume every position empty ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            board(row,col) = EMPTYFIT
        NEXT col
    NEXT row
    REM *** then mark invalid positions ***
    board(1,1) = NOTALLOWED
    board(1,2) = NOTALLOWED
    board(2,1) = NOTALLOWED
    board(2,2) = NOTALLOWED

    board(1,6) = NOTALLOWED
    board(1,7) = NOTALLOWED
    board(2,6) = NOTALLOWED
    board(2,7) = NOTALLOWED
    board(6,1) = NOTALLOWED
    board(6,2) = NOTALLOWED
    board(7,1) = NOTALLOWED
    board(7,2) = NOTALLOWED
    board(6,6) = NOTALLOWED
    board(6,7) = NOTALLOWED
    board(7,6) = NOTALLOWED
    board(7,7) = NOTALLOWED
    REM *** Initialise boardContents ***
    REM *** with the pit coordinates ***
    x = -9
    z = 9

    FOR row = 1 TO 7
        FOR col = 1 TO 7
            BoardCoords(row,col).x = x
            BoardCoords(row,col).z = z
            x = x + 3
        NEXT col
        z = z - 3
        x = -9
    NEXT row
ENDFUNCTION

FUNCTION CreateMarbles()
    REM *** Load marble texture ***
    LOAD IMAGE "laval.bmp",marbleimg
    REM *** Create marbles ***
    objjno = firstmarbleobj
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = 0
                AND ((row <> 4) OR (col <> 4))
                MAKE OBJECT SPHERE objjno,2,20,20
                TEXTURE OBJECT objjno,marbleimg
                POSITION OBJECT objjno,
                    boardCoords(row,col).x,0,
                    boardCoords(row,col).z
                board(row,col) = objjno
                INC objjno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

FUNCTION CreateHiddenPitObjects()
    REM *** Set first hidden object ID ***
    pitobjno = firsthiddenobj
    REM *** Load image used to texture
objects ***
    LOAD IMAGE "black.bmp",5
    REM *** FOR each position on board ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            REM *** IF its a valid position ***
            IF board(row,col)>-1
                REM *** Record hidden object ID
for that pit
                pitObjects(row,col) = pitobjno
                REM *** Make, texture and hide
object ***
                MAKE OBJECT SPHERE pitobjno,2
                TEXTURE OBJECT pitobjno,5
                POSITION OBJECT
pitobjno,boardCoords(row,col).x,0,boardCoords
(row,col).z
                SET OBJECT TRANSPARENCY
pitobjno,1
                REM *** Increment object ID
                INC pitobjno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

FUNCTION HelpSetup()
    REM *** Create the sprite required by
help ***
    CREATE ANIMATED SPRITE
helpsprite,"rules.bmp",3,1,helpimg
    REM *** Position sprite ***
    SPRITE helpsprite,10,10,helpimg
    REM *** Set sprite to show first page ***
    SET SPRITE FRAME helpsprite,1
    REM *** Hide sprite ***
    HIDE SPRITE helpsprite
ENDFUNCTION

```

```

FUNCTION GetPlayersMove()
  REM *** Read keyboard/mouse ***
  REPEAT
    mouse = MOUSECLICK()
    code = SCANCODE()
  UNTIL code <> 0 OR mouse <> 0
  REM *** Assume not quitting ***
  quit = 0
  REM *** Mouse click same as Enter ***
  IF mouse <> 0
    code = 28
  ENDIF
  SELECT code
    CASE 28      `Enter key
      REM *** IF start of turn THEN ***
      IF gamestate = 1
        REM *** Select marble and
        REM *** IF OK, state 2, ***
        IF SelectMarble()
          gamestate = 2
        ENDIF
      REM *** IF second stage of turn ***
      ELSE IF gamestate = 2
        REM *** Select pit and
        REM *** IF OK,
        IF SelectPit()
          REM *** Move marble ***
          MoveMarble()
          REM *** Start new turn ***
          gamestate = 1
        ENDIF
      ENDIF
    ENDCASE
    CASE 22      `U - undo selection
      IF gamestate = 2
        objno =
        board(move(1).row,move(1).col)
        TEXTURE OBJECT objno,marbleimg
        gamestate = 1
      ENDIF
    ENDCASE
    CASE 49      `N - new game
      ScreenSetUp()
      GameSetUp()
    ENDCASE
    CASE 59      `F1 - Help
      IF gamestate < 10
        SelectHelpPage(1)
        gamestate = gamestate + 10
      ELSE
        SelectHelpPage(0)
        gamestate = gamestate - 10
      ENDIF
    ENDCASE
    CASE 37      `K - Help page 2
      IF gamestate > 10
        SelectHelpPage(2)
      ENDIF
    ENDCASE
    CASE 19      `R - Help page 3
      IF gamestate > 10
        SelectHelpPage(3)
      ENDIF
    ENDCASE
    CASE 33      `F - Quit game
      quit = 1
    ENDCASE
  ENDSELECT
  REM *** Wait keys and mouse released ***
  WHILE SCANCODE() <> 0 OR
  MOUSECLICK() <> 0
  ENDWHILE
ENDFUNCTION quit

FUNCTION SelectMarble()
  REM *** Check for marble ***
  x = MOUSEX()
  y = MOUSEY()
  objno = PICK OBJECT
  (x,y,firstmarbleobj,lastmarbleobj)
  REM *** IF no marble ***
  IF objno = 0
    REM *** Show message; return 0
    DisplayMessage(100,100,
    "Choose a marble",2)
    EXITFUNCTION 0
  ENDIF
  REM *** Find marble's ID in board ***
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      IF board(row,col) = objno
        prow = row
        pcol = col
      ENDIF
    NEXT col
  NEXT row
  REM *** Change marble colour ***
  COLOR OBJECT objno,RGB(255,255,0)
  REM *** record position of marble
  move(1).row = prow
  move(1).col = pcol
ENDFUNCTION 1 `Valid marble selected

FUNCTION SelectPit()
  REM *** Check for hidden obj or marble
  x = MOUSEX()
  y = MOUSEY()
  objno = PICK OBJECT
  (x,y,firsthiddenobj,lasthiddenobj)
  marbleno = PICK OBJECT(
  x,y,firstmarbleobj,lastmarbleobj)
  REM *** IF no hidden obj OR marble THEN
  IF objno = 0 OR marbleno <> 0
    REM *** Display message, return 0 ***
    DisplayMessage(100,100,
    "Choose an empty space",2)
    EXITFUNCTION 0
  ENDIF
  REM *** Find pit's ID in pitObjects ***
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      IF pitObjects(row,col) = objno
        prow = row
        pcol = col
      ENDIF
    NEXT col
  NEXT row
  REM *** Record move-to position ***
  move(2).row = prow
  move(2).col = pcol
  REM *** IF not a valid move THEN ***
  IF NOT isValidMove()
    REM *** Display message;return 0 ***
    DisplayMessage(100,100,
    "Invalid move",2)
    EXITFUNCTION 0
  ENDIF
ENDFUNCTION 1 `Valid move-to pit selected

FUNCTION IsValidMove()
  REM *** Invalid if start and finish pits
  REM *** not exactly 2 apart distant ***
  REM *** or no marble in-between ***
  test1 = (ABS(move(1).row-move(2).row)=2)
  AND (move(1).col = move(2).col)
  test2 = (ABS(move(1).col-move(2).col)=2)
  AND (move(1).row = move(2).row)
  midrow = ABS(move(1).row+move(2).row)/2
  midcol = ABS(move(1).col+move(2).col)/2
  test3 = board(midrow,midcol) > 1
  REM *** Return 1 if tests check okay ***
  IF (test1 OR test2) AND test3
    result = 1
  ELSE
    `otherwise return zero
    result = 0
  ENDIF
ENDFUNCTION

```

```

ENDIF
ENDFUNCTION result

FUNCTION MoveMarble()
  REM *** Move marble in board ***
  board(move(2).row,move(2).col) =
  board(move(1).row,move(1).col)
  board(move(1).row,move(1).col) = 0
  REM *** Move marble on screen ***
  POSITION OBJECT
  board(move(2).row,move(2).col),
  boardCoords(move(2).row,move(2).col).x,0,
  boardCoords(move(2).row,move(2).col).z
  TEXTURE OBJECT board(move(2).row,move(2).col),
  marbleimg
  REM *** Calculate position of jumped marble ***
  jumpedrow = (move(1).row+move(2).row)/2
  jumpedcol = (move(1).col+move(2).col)/2
  REM *** Remove jumped marble from screen ***
  jumpedmarble = board(jumpedrow,jumpedcol)
  HIDE OBJECT jumpedmarble
  DELETE OBJECT jumpedmarble
  REM *** Remove jumped marble from board ***
  board(jumpedrow,jumpedcol)=0
  REM *** Decrement marblesremaining ***
  DEC marblesremaining
ENDFUNCTION

FUNCTION SelectHelpPage(pageno)
  IF pageno < 0 OR pageno > 3
    EXITFUNCTION
  ENDIF
  IF pageno > 0
    SET SPRITE FRAME helpsprite,pageno
    SHOW SPRITE helpsprite
  ELSE
    HIDE SPRITE helpsprite
  ENDIF
ENDFUNCTION

FUNCTION DisplayMessage(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y
    PRINT message$
  ENDWHILE
ENDFUNCTION

```

