

The Complete Game

The Game Logic

There are several stages to be completed each time a player takes a turn. These can be summarised as:

```
Throw the dice
Select the piece to be moved
Move the piece
Handle any collisions
```

However, it will not always be possible to move a piece - for example, at the start of the game, if a six is not thrown then no move is allowed.

Also, collisions are fairly infrequent and only occur if the piece being moved ends up on the same square as another piece.

So, we can make our stages more code-like by modifying it to include IF statements:

```
Throw the dice
IF a piece is available for moving THEN
  Select piece to be moved
  Move the piece
  IF the piece ends up on an occupied square THEN
    Handle the collision
  ENDIF
ENDIF
ENDIF
```

To describe the logic of the complete game, all we need to do is add; game initialisation, use of the slider to change the viewpoint, the ability to move on to the next player when a turn is complete, and a check for the game being won. This gives us the following overall game logic:

```
Set up screen
Set up game
REPEAT
  Allow for change of viewpoint
  Throw the dice
  IF a piece is available for moving THEN
    Select piece to be moved
    Move the piece
    IF the piece ends up on an occupied square THEN
      Handle the collision
    ENDIF
  ENDIF
  Change to next player
UNTIL game won
Close game
```

Although the logic seems fine, actually it doesn't play very well because the player will find that it's almost impossible to drag the viewpoint slider before the logic has moved on to the dice throw part. To fix this we need to make the dice throw optional (just like using the slider). Of course, the next step in the logic can only be performed once the dice throw has been made. This means that the program continually iterates, waiting for either the slider or dice being activated. Now the logic of the main loop becomes:

```
Allow for change of viewpoint
Allow dice throw
```

```

IF dice throw made THEN
  IF a piece is available for moving THEN
    Select piece to be moved
    Move the piece
    IF the piece ends up on an occupied square THEN
      Handle the collision
    ENDIF
  ENDIF
ENDIF
Change to next player
ENDIF

```

If we are to perform a different part of the logic each time the loop is iterated, the usual way to achieve this is to create a **state** variable.

A state variable is used to record which part of the logic we are about to perform. For example we can set it to 0 to indicate that we are waiting for the dice to be thrown, 1 to indicate that we are waiting for the piece to be moved, 2 to indicate that a piece is to be selected and so on, covering each of the main stages involved in a single turn.

So at last we have the code for the main logic of our program:

```

REM *****
REM ***   Main Game Logic   ***
REM *****
SetUpScreen()
SetUpGame()
state = 0
REPEAT
  REPEAT
    HandleSlider()
  SELECT state
    CASE 0 `Allow dice throw
      throw = HandleDice()
      IF throw <> 0
        state = 1
      ENDIF
    ENDCASE
    CASE 1 `Select available piece
      IF pieceAvailable(throw)
        p = SelectPiece(throw)
        state = 2
      ELSE
        state = 5 `Move to next player
      ENDIF
    ENDCASE
    CASE 2 `Move selected piece
      bumpID = HandlePiece(playerToMove,p,throw)
      IF bumpID <> 0
        state = 3 `Handle collision
      ELSE
        state = 4 `Check for game over
      ENDIF
    ENDCASE
    CASE 3 `Handle collision
      HandleCollision(playerToMove,p,bumpID,throw)
      state = 4
    ENDCASE
    CASE 4 `Check for game over
      gameOver = CheckForGameOver()
      IF gameOver
        state = 6
      ELSE
        IF throw = 6 `If 6 thrown player throws again
          state = 0
        ENDIF
      ENDIF
    ENDCASE
  ENDREPEAT
ENDREPEAT

```

```

        ELSE
            state = 5
        ENDIF
    ENDIF
ENDCASE
CASE 5 `Move on to next player
    PlayerToMove = PlayerToMove mod 4 + 1
    SET SPRITE FRAME playerSpr,PlayerToMove
    state = 0
ENDCASE
CASE 6
    `Game over
ENDCASE
ENDSELECT
UNTIL state = 6
WAIT KEY
END

```

In addition, the code requires all the other programs we created earlier, as well as named constants and global variables, so the above code needs to be preceded by the following lines:

```

#include "PointType.dba"
#include "SliderType.dba"
#include "Point3DType.dba"
#include "LudoPositionType.dba"
#include "LudoPieceType.dba"

REM *****
REM ***   Named Constants   ***
REM *****
REM *** 3D Objects ****
#CONSTANT boardObj      1
#CONSTANT diceObj      2
#CONSTANT firstRedPiece 3
#CONSTANT firstBluePiece 7
#CONSTANT firstYellowPiece 11
#CONSTANT firstGreenPiece 15
REM *** Images ***
#CONSTANT sliderImg     1
#CONSTANT thumbImg     2
#CONSTANT pointerImg    3
#CONSTANT frameImg     4
#CONSTANT playerImg    5
REM *** Sprites ***
#CONSTANT pointerSpr    3
#CONSTANT frameSpr     4
#CONSTANT playerSpr    5

REM *****
REM ***   Global Variables   ***
REM *****
GLOBAL memblockID = 0
GLOBAL DIM pieces(4,4) AS LudoPieceType
GLOBAL DIM home(4,4) AS LudoPositionType
GLOBAL DIM board(52) AS LudoPositionType
GLOBAL DIM stairs(4,5) AS LudoPositionType
GLOBAL DIM goal(4,4) AS LudoPositionType
GLOBAL slider AS SliderType
GLOBAL playerToMove = 1

```

Activity 065
 Modify your own main section to match the code given above.

Level 1 Routines

Now we're ready to write each of the routines called by the main program logic. Three of these we completed earlier: *SetUpScreen()*, *SetUpGame()* and *HandleSlider()* so we'll start here with *HandleDice()*.

HandleDice()

This routine checks to see if the mouse has been clicked over the dice image. If it has, then the dice is rolled and the value generated returned; if the mouse click has not been made, a value of zero is returned. The code is:

```
FUNCTION HandleDice()
SET CURRENT CAMERA 2
REM *** If mouse clicked over dice, roll dice ***
IF MOUSECLICK() = 1
    IF PICK OBJECT(MOUSEX(), MOUSEY(), diceObj,diceObj)
        result = RollDice()
    ELSE
        result = 0
    ENDIF
ELSE
    result = 0
ENDIF
REM *** Wait until mouse button no longer down ***
WHILE MOUSECLICK() <> 0
    ENDWHILE
ENDFUNCTION result
```

PieceAvailable()

The routine returns 1 if a piece is available for moving and zero if no piece is available.

If a piece has already left its home area, then it can be moved after the dice is thrown, but if no pieces are in the playing area, then a piece can be moved from the home area onto the playing area only if a 6 has been thrown on the dice.

Because the returned value can depend on the value thrown on the dice, that value (*t*), is passed as a parameter to this routine.

The code is:

```
FUNCTION PieceAvailable(t)
    result = 1
    total = AtHome() + AtGoal()
    IF total=4 AND t <> 6
        result = 0
    ENDIF
ENDFUNCTION result
```

Notice that the routine calls two other routines: *AtHome()* and *AtGoal()* which return the current player's number of pieces in the home area and in the goal area respectively. These new routines will be coded as level 2 routines.

SelectPiece()

This routine allows the player to select which piece is to be moved.

The code is quite complicated because it needs to determine which pieces belong to the current player (so that other players' pieces cannot be selected) and which of those pieces can actually be moved (pieces that have reached the goal area cannot be moved and pieces in the home area cannot be moved unless a 6 was thrown).

This routine takes the dice throw (*t*) as a parameter.

The routine returns which piece has been selected. This value is in the range 1 to 4 representing which of the player's four pieces has been selected.

```
FUNCTION SelectPiece(t)
  SET CURRENT CAMERA 0
  REPEAT
    REM *** Calculate the 3D object ID range for ***
    REM *** this player's pieces ***
    firstID = (playerToMove-1)*4 + 3
    lastID = firstID + 3
    REM *** Wait for click on piece ***
    REPEAT
      IF MOUSECLICK()
        id = PICK OBJECT(MOUSEX(),MOUSEY(),firstID,lastID)
      ENDIF
    UNTIL id <> 0
    REM *** Convert ID to r, c values ***
    rc = FindPiece(id)
    r = rc / 10
    c = rc mod 10
    REM *** IF selected piece in goal area, reject it ***
    IF IsOnSteps(pieces(r,c))=6
      id = 0
    ENDIF
  UNTIL id <> 0 AND GetSquaresMoved(pieces(playerToMove,c))
  ⤴ <> 0 OR t = 6
  REM *** Wait until mouse unclicked ***
  WHILE MOUSECLICK() <> 0
  ENDWHILE
ENDFUNCTION post
```

Secondary functions used here are:

<i>FindPiece()</i>	which converts the 3D ID value to an <i>rc</i> value.
<i>IsOnSteps()</i>	which returns which of the steps leading to the goal area a piece is on.
<i>GetSquaresMoved()</i>	which returns the number of squares moved by a specified piece.

The above routines are level 2.

HandlePiece()

This routine moves the selected piece one square at a time around the board. It has to check for the piece being on the stairs since that situation requires use of different variables than when on the main board. Also if the piece arrives at the goal area it must be positioned correctly and any additional square moves still outstanding are ignored. The routine also updates the data in the main data arrays as necessary.

The routine takes the id value of the piece (*r* and *c*) and the dice throw value (*t*) as parameters.

The routine returns details of any collision that has occurred by moving the piece.

The code is:

```

FUNCTION HandlePiece(r,c,t)
  REM *** Get number of squares already moved by piece ***
  oldpost = GetSquaresMoved(pieces(r,c))
  REM *** Get piece's 3D id ***
  id = GetPieceModelID(pieces(r,c))
  REM *** Move piece one square at a time ***
  FOR x = 1 TO t
    result = 0
    MovePieceOneSquare(pieces(r,c))
    REM *** Calculate piece's position on board ***
    square = (GetSquaresMoved(pieces(r,c))+
    ↵ GetStartSquare(pieces(r,c))-2) mod 52 + 1
    REM *** Determine if piece on steps ***
    p = IsOnSteps(pieces(r,c))
    REM *** IF not on steps THEN ***
    IF NOT p
      REM *** Position piece normally ***
      POSITION OBJECT id,GetPositionX(board(square)),
      ↵ GetPositionY(board(square)),
      ↵ GetPositionZ(board(square))
    ELSE
      REM *** IF in main stair area THEN ***
      IF p < 6
        REM *** Position on stair area ***
        POSITION OBJECT id,GetPositionX(stairs(r,p)),
        ↵ GetPositionY(stairs(r,p)),
        ↵ GetPositionZ(stairs(r,p))
      ELSE
        REM *** Position in goal area ***
        POSITION OBJECT id,GetPositionX(goal(r,c)),
        ↵ GetPositionY(goal(r,c)),GetPositionZ(goal(r,c))
      ENDIF
    ENDIF
    WAIT 10
  NEXT x
  REM *** Calculate new number of squares moved ***
  newpost = oldpost + t
  REM *** Update data structures ***
  result = UpdateBoard(r,c,oldpost,newpost)
ENDFUNCTION result

```

This time the new level 2 routines called are:

- GetPieceModelID()* which returns the 3D id of a model.
- MovePieceOneSquare()* which moves a piece one square along the board.
- GetStartSquare()* which returns the square on which a piece started its journey around the board.
- GetPositionX()* which returns a board position's x-ordinate.
- GetPositionY()* which returns a board position's y-ordinate.
- GetPositionZ()* which returns a board position's z-ordinate.

UpdateBoard()

which updates the data structures when a piece is moved.

HandleCollision()

This routine handles collisions between two pieces when the moved piece lands on an already occupied square. There are two separate possibilities in such a situation. The first of these is that the piece already on the square is of a different colour from the moved piece and in this case that stationary piece must be returned to its home position. The second possibility is that the two pieces involved are of the same colour; in this case the moved piece is returned to its previous position and the move is forfeited.

The routine takes as parameters the current player (*r*) and the number of the piece (*c*), the 3D id value of the piece already on the square (*bumpID*), and the value thrown on the dice (*t*).

The code for the routine is:

```
FUNCTION HandleCollision(col, no, bumpID, t)
  REM *** Convert 3D id to r,c value ***
  rc = FindPiece(bumpID)
  r = rc / 10
  c = rc mod 10
  REM *** IF the pieces are the same colour THEN ***
  IF GetPieceColour(pieces(col,no)) =
  GetPieceColour(pieces(r,c))
    REM *** Return moved piece to last position ***
    ReturnMovedPiece(col,no,t)
  ELSE
    REM *** Move existing piece to home ***
    HomePiece(r,c)
  ENDF
ENDFUNCTION
```

The routine calls only three new functions:

<i>FindPiece()</i>	which returns the <i>rc</i> value of a piece as a single value.
<i>GetPieceColour()</i>	which returns the colour of a specified piece.
<i>ReturnMovedPiece()</i>	which returns a specified piece to its previous position.
<i>HomePiece()</i>	which returns a specified piece to its home position.

CheckForGameOver()

The final routine called by the main logic checks to see if the current player has all four pieces in the goal area.

The routine returns 1 if all four pieces are in the goal area, otherwise it returns zero.

The code for the routine is:

```
FUNCTION CheckForGameOver()  
    result = AtGoal()=4  
ENDFUNCTION result
```

Note that the routine does nothing more than call the *AtGoal()* function to perform its task.

Activity 055

Enter the code for all level 1 routines into your program.