Dots and Boxes Tcl/Tk GUI Documentation

Eudean Sun

2006-12-18

Contents

1 Introduction

This document outlines the methodology and techniques used to develop the Tcl/Tk GUI for the game Dots and Boxes, part of the Gamesman system. It is meant as a companion to the source code for understanding the logic behind some of the more complicated procedures in the source. It can also be seen as a basic tutorial for structuring and developing a Tcl/Tk GUI for games within the Gamesman system as part of the "Gold" interface standard, using Dots and Boxes as the basis of its examples.

This file refers to `~/tcl/mdnb.tcl` almost exclusively. Minor discussion of the features of `~/src/mdnb.cc` are included, but any discussion of `~/bin/Xmdnb` and `~/bin/XGamesman` is excluded (these should be documented independently since they are the same for all games).

2 Globals

I defined a handful of global variables at the top of the `mdnb.tcl` to ease the development of the GUI. They are as follows:

- `boardWidth` (default: 2): current width of the board (in number of boxes)
- `boardHeight` (default: 2): current height of the board (in number of boxes)
- `margin` (default: 100): margin around the board (in pixels)
- `maxWidth` (default: 3): maximum board width (in number of boxes)
- `maxHeight` (default: 3): maximum board height (in numbmer of boxes)
- `r` (default: 5): radius of dots on the board (in pixels)
- `basespeed` (default: 50): base speed of the game (in milliseconds approximately)
- `turn` (default: 0): a flag tracking whose turn it is

These will be described in more detail in the discussion of specific procedures. A couple are meant for setting once and simply referencing them (`margin`, `maxWidth`, `maxHeight`, `r`, `basespeed`, `turn`). `boardWidth` and `boardHeight` are changed when we change variants, so we must keep those properly updated when the player uses the Rules menu.

There are some globals defined later in the code, for example `square`, which will be discussed where they are defined.

3 Drawing the Board

When drawing a grid of squares, the first thing I do is compute the length of the side of the square. This is an extremely useful value to have when drawing or placing any other element, so doing it first makes sense. I compute this length by checking which is larger, the `boardWidth` or the `boardHeight`. I take the larger side and divide the length of the board in that direction (in pixels, minus the margins on each side) by the size of the board in that direction (in number of boxes). Here is the code that computes this value, which I call `square`:

```
if {$boardHeight > $boardWidth} {
    set square [expr ($gFrameHeight-2*$margin) / $boardHeight]
} else {
    set square [expr ($gFrameWidth-2*$margin) / $boardWidth]
}
```

I declare `square` as a global at the top of the function I use to draw the board, which I call `DrawBoard`. Now, to draw the board, I simply have to draw a grid of dots (that is `boardWidth` + 1 by `boardHeight` + 1) that are separated by a distance `square` in the horizontal and vertical directions. The reason the dots extend to `boardWidth` + 1 and `boardHeight` + 1 is because I define the width and height of the board to be the number of boxes created by drawing the grid, which means there is one extra edge in each direction. Here is the code to draw this grid:

```
for {set i 0} {$i < $boardHeight+1} {incr i} {
    for {set j 0} {$j < $boardWidth+1} {incr j} {
      $c create oval \
          [expr ($margin+$j*$square)-$r] \
          [expr ($margin+$i*$square)-$r] \
          [expr ($margin+$j*$square)+$r] \
          [expr ($margin+$i*$square)+$r] -fill black
    }
}
```

We're almost done. If you were to run the above code, you'd see the board, but you'd also see the Gamesman background bleeding through it. We have to give the game some type of background. Typically, a gray square is perfectly sufficient, and that is what you'll see many games use. However, since Dots and Boxes is typically played on lined sheets of paper, we can make the background a little more interesting. First, we need to draw a white background to represent the paper. We then put a vertical red stripe and tile horizontal blue lines to complete the effect. The following lines of code accomplish this (the placement of elements was done by sight to match a piece of paper):

```
$c create rect -1 -1 $gFrameWidth $gFrameHeight -fill white
$c create line [expr $margin-10] 0 [expr $margin-10] $gFrameHeight \
    -fill red -width 1

for {set i [expr $margin-10]} {$i < $gFrameHeight} {set i [expr $i+20]} {
    $c create line 0 $i $gFrameWidth $i -fill blue -width 1
}
```

With that, we are done drawing the board.

4 Drawing the Pieces

Now, we need to be able to populate the blank board with pieces. In this case, pieces refer simply to lines that connect the dots either horizontally or vertically. To do this, we write a procedure called `DrawPieces`, which takes in the canvas and a position to draw. We must first unhash the position, then iterate over the resulting board, drawing a piece at every location on that board. Here is the implementation of `DrawPieces`:

```
proc DrawPieces { c position } {
    global boardWidth boardHeight
    set board [unhash $position]
    for {set i 0} \
        {$i < [expr $boardWidth*($boardHeight+1)+$boardHeight*($boardWidth+1)]} \
        {incr i} {
        if {[string index $board $i] == 1} {
            DrawPiece $c $i
        }
    }
}
```

Now we'll take a look at unhashing and the `DrawPiece` procedure.

4.1 Unhashing the Position

To write the `unhash` procedure, I simply looked at mdnb.cc and figured out how the C code goes about unhashing positions in the `Print` procedure of its `DNB` struct. Since the C code is a bit complex, I won't detail how I figured out the unhashing procedure exactly. In typical examples, unhashing in Tcl will be a matter of either converting some basic C code into Tcl, or calling `C_GenericUnhash`. Here is the implementation of `unhash`:

```
proc unhash { position } {
    global boardWidth boardHeight
    for {set i 0} \
        {$i < [expr $boardWidth*($boardHeight+1)+$boardHeight*($boardWidth+1)]} \
        {incr i} {
        if {[expr (1<<$i) & $position] != 0} {
            append board 1
        } else {
            append board 0
        }
    }
    return $board
}
```

Your unhashing procedure should almost always take in a position and return a string (or list) of characters representing pieces on the board. In this case, since there's only one type of piece (a line) plus blank pieces, our string is simply binary. In many other cases (for example, Tic-Tac-Toe, Asalto, and Achi), you'll have some ternary representation, one character for blanks, another for one type of piece, and the third for the other type of piece.

4.2 Drawing a Single Piece

To build up the ability to draw a board full of pieces, I usually start with the ability to draw just one piece. This procedure typically takes in either an index into the board, or a row and a column (if you take in an index, you will often compute a row and column to make your life easier anyway, so neither choice carries a big advantage). This procedure is very specific to each game—for example, some games draw X's and O's in a square grid (e.g. Tic-Tac-Toe), some draw circles in strange grids (e.g. Asalto), etc.—and in this case we're drawing lines between dots.

The first thing I establish is whether I'm going to draw a vertical or horizontal line. This is because the way the moves are numbered, it is easy to determine whether it is a horizontal or vertical line. The moves are numbered starting from 0 at the top left horizontal line, incrementing right first, then down on the horizontal lines. Then come the vertical lines, which are numbered starting from `boardWidth` · (`boardHeight` + 1) at the top left vertical line, then incrementing down first, then right.

Here's a simple text representation of a board labeling the moves in a 2 by 2 board:

```
 -----------
|  0  |  1  |
|6    |8    |10
|_____|_____|
|  2  |  3  |
|7    |9    |11
|_____|_____|
   4     5
```

From here you can see the clear partition between horizontal and vertical lines in moves. It should be no problem to figure out the math to draw a move at a given index. For the horizontal lines, we figure out what row we're in, then draw a line in that row from the column left of the move to the column right of the move. We perform a similar computation for the vertical lines. Here is the code that performs this task:

```
proc DrawPiece { c index } {
    global margin square boardWidth boardHeight

    if {$index < [expr ($boardHeight+1)*$boardWidth]} {
        $c create line \
            [expr $margin+($index%$boardWidth)*$square] \
            [expr $margin+($index/$boardWidth)*$square] \
            [expr $margin+($index%$boardWidth+1)*$square] \
            [expr $margin+($index/$boardWidth)*$square] \
            -fill black -width 4 -tag "pieces"
    } else {
        set index [expr $index-($boardHeight+1)*$boardWidth]
        $c create line \
            [expr $margin+($index/$boardHeight)*$square] \
            [expr $margin+($index%$boardHeight)*$square] \
            [expr $margin+($index/$boardHeight)*$square] \
            [expr $margin+($index%$boardHeight+1)*$square] \
            -fill black -width 4 -tag "pieces"
    }
}
```

4.3 Putting It All Together: GS_DrawPosition

Finally, having implemented `DrawPieces`, we can implement `GS_DrawPosition`, which basically just calls our DrawPieces procedure. Before doing so, however, it deletes the pieces currently on the board. This ensures we start with a clean board when drawing the position. Its implementation is as follows:

```
proc GS_DrawPosition { c position } {
    $c delete pieces
    DrawPieces $c $position
}
```

5 Showing Moves

5.1 GS_ShowMoves

The implementation of `GS_ShowMoves` is extremely simple. First, we clear the board of moves that we drew previously. Then, we iterate through the `moveList` and draw each move individually by calling a procedure called `DrawMove`.

```
proc GS_ShowMoves { c moveType position moveList } {
    $c delete moves
    foreach move $moveList {
        DrawMove $c [lindex $move 0] $moveType [lindex $move 1]
    }
}
```

5.2 Drawing a Move

The procedure `DrawMove` takes in a move, a move type (i.e. value move or not), and the type of move that it is based on the move type (i.e. win, tie, or loss). The first step in drawing a move is figuring out what to color the move based on the `moveType` setting and what value the move is. If `moveType` is value moves, then we have to color based on whether the move is a win (dark red), loss (green), or tie (yellow). Recall that the value of the move specifies what you're handing to the other player, which is why a win is dark red and not green (you're giving the other player a winning position).

After establishing the appropriate color, we simply have to draw the move on the board. In this case, the move is just an index into the board, so we have to draw the vertical or horizontal line associated with that index. Incidentally, this is just like drawing a piece. This will often be the case, since denoting moves often involves putting a smaller, cyan piece where a normal piece would normally go. In this case, we can re-use the same logic as we used in the `DrawPiece` procedure to draw the moves. We set the drawn move to a temporary variable so that we can easily bind it using the last line in the procedure. The prior two lines are used to turn the move black when moused over and back to its original color when the mouse is moved off the piece.

```
proc DrawMove { c move moveType type } {

    global margin square boardWidth boardHeight r

    switch $moveType {
        value {
            switch $type {
                Win { set color darkred }
                Lose { set color green }
                Tie { set color yellow }
                default { set color cyan }
            }
        }
        default {
            set color cyan
        }
    }

    if {$move < [expr ($boardHeight+1)*$boardWidth]} {
        set tmp [$c create line \
            [expr $margin+($move%$boardWidth)*$square+$r] \
            [expr $margin+($move/$boardWidth)*$square] \
            [expr $margin+($move%$boardWidth+1)*$square-$r] \
            [expr $margin+($move/$boardWidth)*$square] \
            -fill $color -width $r -tag "moves"]
    } else {
        set tmp [$c create line \
            [expr $margin+(($move-($boardHeight+1)*$boardWidth)/$boardHeight)*$square] \
            [expr $margin+(($move-($boardHeight+1)*$boardWidth)%$boardHeight)*$square+$r] \
            [expr $margin+(($move-($boardHeight+1)*$boardWidth)/$boardHeight)*$square] \
            [expr $margin+(($move-($boardHeight+1)*$boardWidth)%$boardHeight+1)*$square-$r] \
            -fill $color -width $r -tag "moves"]
    }

    $c bind $tmp <Enter> "$c itemconfigure $tmp -fill black"
    $c bind $tmp <Leave> "$c itemconfigure $tmp -fill $color"
    $c bind $tmp <ButtonRelease-1> "ReturnFromHumanMove $move"
}
```

6 Handling Moves

There are three parts to handling moves, one of which we've already covered, which is drawing the new position. The other parts are animating the move and tracking box owners.

6.1 Animation

The first thing we do when handling a move is animate it (though the animation code is typically the last to be written—make sure `GS_DrawPosition` works first!). As soon as the animation is finished, we can call `GS_DrawPosition` on the new position to ensure the board is completely up-to-date after animating. To do the animation, though, we'll need to figure out what what line we're drawing and what direction we're drawing it in. The latter is rather simple—I decided that for a right-handed person, it is most natural to draw horizontal lines left-to-right and vertical lines top-to-bottom, so our implementation of Dots and Boxes follows this convention. As for the former, that requires a little math, which I'll explain shortly.

First, we should set up some basic variables to help us do the animation. Here is some code to do that:

```
set speed [expr $basespeed / pow(2, $gAnimationSpeed)]
set delta [expr $square / $speed]

set dir 0
set col [expr $move%$boardWidth]
set row [expr floor($move/$boardWidth)]
if {$move >= [expr ($boardHeight+1)*$boardWidth]} {
    set move [expr ($move-($boardHeight+1)*$boardWidth)]
    set row [expr $move%$boardHeight]
    set col [expr floor($move/$boardHeight)]
    set dir 1
}
```

The `speed` variable represents the speed, in milliseconds, of our animation. `basespeed` defaults to 50, so if the user does not touch the animation slider, the animation will take about 50ms. `speed` scales exponentially with `gAnimationSpeed`, so if `gAnimationSpeed` is set to 2, `speed` is halved, so the animation is twice as fast. Likewise if `gAnimationSpeed` is set to -1, then `speed` doubles to 100ms.

Using `speed`, we can then compute `delta`, which represents a differential unit we have to increment our animated piece by to complete the animation in the time defined by `speed`. Naturally, `delta` is just the total distance the object has to travel divided by `speed`. In this case, our lines are always between two dots either horizontally or vertically, meaning the lines will always be of length `square`. Hence, we take `square`, divide by `speed`, and that value is assigned to `delta`.

Next, we try to discover in what direction, row, and column we'll be animating. We default to horizontal (direction 0), and compute the row and column accordingly (i.e. assuming that `move` refers to a horizontal line). We check if our assumption is correct, and if not, we re-define `row`, `col`, and `dir` based on a vertical move. Now that we know where and in what direction we need to animate the line, we can begin animating it.

Animation naturally always takes place in a for loop. This is exactly why we computed `delta`! We can iterate over a variable ranging from 0 to `square` in steps of `delta`. Our strategy for animation is simple: draw the state at the current point in time and tag it with something unique, then before drawing it subsequent times, delete elements with that unique tag. The time advances in steps of approximately 1ms. We check what direction we're animating in, then use the row and column we computed earlier to figure our the starting point of the line. The index of iteration for the for loop then handles animating the line. Here is the animation code:

```
for {set i 0} {$i < $square} {set i [expr $i+$delta]} {
    if { $dir == 0 } {
        $c create line \
            [expr $margin+$col*($square)] \
            [expr $margin+$row*($square)] \
            [expr $margin+$col*($square)+$i] \
            [expr $margin+$row*($square)] \
            -width 4 -fill black -tag tmp
    } else {
```

```
        $c create line \
            [expr $margin+$col*($square)] \
            [expr $margin+$row*($square)] \
            [expr $margin+$col*($square)] \
            [expr $margin+$row*($square)+$i] \
            -width 4 -fill black -tag tmp
    }
    after 1
    update idletasks
    $c delete tmp
}
```

6.2 Tracking Box Owners

We must also track who won what box in `mdnb.tcl`. Although the C code in `mdnb.cc` does provide this capability through a variable called `TrackBoxOwners`, the C functionality does not scale beyond 2 by 3 boards. Hence, we ignore the tracking of box owners within the hash function and simply track it via some code in the Tcl/Tk GUI. This allows it to scale to 3 by 3, which is a highly desirable board size to play on versus the more trivial 2 by 2 and 2 by 3 boards.

6.2.1 GS_HandleMove

We must modify `GS_HandleMove` in order to support tracking box owners. When a move is made, we must check if a box was completed on that move, and if so, label it accordingly. This is the code that tracks box owners:

```
set count 0
for {set i 0} {$i < $boardWidth * $boardHeight} {incr i} {
    set box [BoxCompleted [unhash $newPosition] $i $theMove]

    if { $box != -1 } {
        LabelBox $c [lindex $box 0] [lindex $box 1] $turn
        incr count
        if {$count == 2} {
        break
    }
}

if { $count == 0 } {
    ChangeTurn
}
```

We first initialize a variable called `count` to zero. Whenever we find a box that was completed on the last turn, we increment count. This is important to ensure player turns are tracked appropriately. If you recall earlier I described a flag called `turn` that keeps track of whose turn it is. When a move is made, if a box was not completed, we call a function called `ChangeTurn` that changes whose turn it is (its implementation is omitted from this document since it is so trivial). If the move did end up completing a box, then we don't change `turn`. In doing this, we are able to track whose turn it is—now all we have to do is detect when a turn ends up completing a box, and color that box according to whose turn it was when it was completed using `LabelBox`.

Thus, we need to write the function `BoxCompleted`, which takes in a board, an index, and a move, then returns the row and column of the box that was completed by that move. We begin by figuring out which row and column the move were in and use those to calculate the top, bottom, left, and right sides of the box that we're checking (`half` is a helper variable in computing the left and right sides of the box, since those are vertical). Once we have these variables defined, we first check to see whether the move was related to

the box that we're checking. If not, then we return −1, since the box was not completed by the move. If it is related, we check the board to see whether the top, bottom, left, and right sides of the box are done, and if so, we've found a box, so we return the row and column. The code for `BoxCompleted` follows.

```
proc BoxCompleted { board index move } {
    global boardWidth boardHeight
    # we have boardWidth * boardHeight boxes
    # for the nth box, check the following sides:
    #
    # row = n / boardWidth
    # col = n % boardWidth
    # half = boardWidth * (boardHeight + 1)
    #
    # sides are:
    # top: row * boardWidth + col
    # bot: (row+1) * boardWidth + col
    # lef: half + col * boardHeight + row
    # rig: half + (col+1) * boardHeight + row
    set row [expr int($index/$boardWidth)]
    set col [expr $index%$boardWidth]
    set half [expr $boardWidth * ($boardHeight+1)]

    set top [expr int($row * $boardWidth + $col)]
    set bot [expr int(($row+1) * $boardWidth + $col)]
    set lef [expr int($half + $col * $boardHeight + $row)]
    set rig [expr int($half + ($col+1) * $boardHeight + $row)]

    # Is the move related to the box?
    if { $top != $move && $bot != $move && $lef != $move && $rig != $move } {
        return -1
    }
    if { [string index $board $top] == 1 && \
         [string index $board $bot] == 1 && \
         [string index $board $lef] == 1 && \
         [string index $board $rig] == 1 } {
        # We've completed a box! Return i, j
        return [list $row $col]
    }
    return -1
}
```

Taking a look back at the code to track box owners, we see that it simply iterates over the entire board and calls `BoxCompleted` on each index with the most recently made move. Although this isn't computationally efficient compared to other methods, for reasonable board sizes it suffices and significantly simplifies the logic of the code.

The last procedure we have to define is `LabelBox`. It's implementation is quite straightforward—it draws a diagonal line through a box given a row and column. It tags line with a general "label"(so that we may delete it when making a new game) and a more specific "l¡row¿¡col¿" so we may delete it when we undo a move.

```
proc LabelBox { c i j flag } {
    global margin square
    if {$flag == -1} {
        return
    } elseif {$flag == 0} {
```

```
        set color "blue"
    } else {
        set color "red"
    }

    # Draw a diagonal line through the box with the appropriate color
    $c create line \
        [expr $margin+$j*($square)] \
        [expr $margin+$i*($square)] \
        [expr $margin+($j+1)*($square)] \
        [expr $margin+($i+1)*($square)] \
        -width 4 -fill $color -tags [list label "l$i$j"]
}
```

6.2.2 GS_HandleUndo

Since we handled tracking box owners in `GS_HandleMove`, we have to keep that information up-to-date when we undo a move as well, meaning we must change `GS_HandleUndo`. The code will look almost the same as the code in `GS_HandleMove`, except instead of drawing a label in the completed box, we're deleting the label that was already there. The implementation follows.

```
proc GS_HandleUndo { c currentPosition theMoveToUndo positionAfterUndo} {

    global boardWidth boardHeight

    set count 0
    for {set i 0} {$i < $boardWidth * $boardHeight} {incr i} {
        set box [BoxCompleted [unhash $currentPosition] $i $theMoveToUndo]

        if { $box != -1 } {
            $c delete "l[lindex $box 0][lindex $box 1]"
            incr count
            if {$count == 2} {
                break
            }
        }
    }

    if { $count == 0 } {
        ChangeTurn
    }

    GS_DrawPosition $c $positionAfterUndo
}
```

7 Variants

In addition to the base functionality outlined above, we'd like to allow the user to change the width and height of the board. This functionality is implemented through `GS_SetupRulesFrame`, `GS_GetOption`, and `GS_SetOption`.

7.1 The Rules Frame

The rules frame is the window that comes up when the user hits the "Rules" button in the Tcl/Tk GUI. Extending `GS_SetupRulesFrame` to include other variants is simple. We setup a new list for every parameter

we want adjustable containing the parameter description and the possible settings for that parameter. We add the additional lists to the `ruleset` list, setup some global variables (`boardWidthOpt` and `boardHeightOpt`) to keep track of the variant settings, and add the names of those variables to the `ruleSettingGlobalNames` list.

The reason we need `boardWidthOpt` and `boardHeightOpt` (instead of just using `boardWidth` and `boardHeight`) is because the globals here take on the value of the index into the list of options that the parameter can take. So, for example, when the user selects "2" as the width of the board, `boardWidthOpt` will be set to "1" instead (the list is zero-indexed). Since we'd prefer to deal with the actual board width and not the index into the list, we keep these values separately and simply use the `Opt` versions of `boardWidth` and `boardHeight` to update the actual width and height variables.

```
set widthRule \
    [list \
        "Select a board width:" \
        "1" \
        "2" \
        "3" \
    ]

set heightRule \
    [list \
        "Select a board height:" \
        "1" \
        "2" \
        "3" \
    ]

# List of all rules, in some order
set ruleset [list $standardRule $widthRule $heightRule]

# Declare and initialize rule globals
global gMisereGame boardWidth boardHeight boardWidthOpt boardHeightOpt
set gMisereGame    0
set boardWidthOpt  1
set boardHeightOpt 1

# List of all rule globals, in same order as rule list
set ruleSettingGlobalNames [list "gMisereGame" "boardWidthOpt" "boardHeightOpt"]
```

7.2 GS_GetOption

This procedure simply returns the option as specified by the C procedure of the same name. We simply convert the C into Tcl and the result is as follows:

```
proc GS_GetOption { } {
    global gMisereGame boardWidth boardHeight maxWidth maxHeight boardWidthOpt boardHeightOpt
    set boardWidth [expr $boardWidthOpt + 1]
    set boardHeight [expr $boardHeightOpt + 1]

    set option 1
    set option [expr $option + $gMisereGame]
    set option [expr $option + 2 * ($boardWidth - 1)]
    set option [expr $option + 2 * $maxWidth * ($boardHeight - 1)]

    return $option
```

```
}
```

7.3 GS_SetOption

This procedure sets the options also in the same manner as the C procedure by the same name. Again, the conversion from C to Tcl is simple and produces the following:

```
proc GS_SetOption { option } {
    global gMisereGame boardWidth boardHeight maxWidth maxHeight

    incr option -1
    set gMisereGame [expr $option%2]
    set boardWidth  [expr $option/2 % $maxWidth + 1]
    set boardHeight [expr $option/(2*$maxWidth) % $maxHeight + 1]
}
```

8 Conclusion

This concludes the guide to the Tcl/Tk GUI of Dots and Boxes. There are a few minor procedures that are self-explanatory and implemented in a straightforward fashion in the code (e.g. `GS_NewGame` and `GS_HideMoves`). For details on those, simply check `mdnb.tcl`. A lot of freedom is available in customizing many of these functions (for example, we could make `GS_HandleUndo` animate if we wanted), so the outline of Dots and Boxes should only be a starting point for customizing the game you are working on.