Abstract

The goal of this document is first to present the Turing Tumble game. This game aims at showing how processors may work "physically" (the signals are modeled via gravity by balls falling down through a vertical boardgame). This game has some limitations, due to its limited size and its fragility (young people playing with it may destroy it very easily.) To overcome these drawback, we would like to put this game back to a computer point of view. Then, to propose a "cahier des charges" to design a computer simulation of it.

1 BoardGame

The boardgame can be seen as a $n \times m$ matrix (n rows, m columns) with $m = 2k + 1$ odd and $n = 2k'$ is even, plus one extra "cell" (precised below). Moreover, two "cells" (two integers $1 \leq b_{init} \leq k$ and $k + 1 < r_{init} \leq 2k + 1$) of the first row must be specified: the ones where the balls (blue b_{init} on the right and red r_{init} on the left) will fall. In the concrete game, $m = 11, n = 10, b_{init} = 4$ and $r_{init} = 8$. Here, we consider the boardgame as a matrix with indices from 1 to n and from 1 to m. The cell (i, j) is the one at the i^{th} row and j^{th} column. The extra cell is the one with coordinates $(n + 1, k + 1)$ (under the last row, in the middle column). Said differently, we may have n to be odd but only the central column of the last row is available (actually, it could be different, here we only mimic the real game that has this solution for reason of space).

As we will see, a ball arriving in a cell of coordinate (i, j), it can go either to the cell $(i-1, j+1)$ or to the cell $(i+1, j+1)$. Hence, when arriving initially in cell $(b_{init}, 1)$, a blue ball can only reach (after $n-1$ falls) a column with index j with $b_{init} - n + 1 \leq j \leq b_{init} + n - 1$ and a red ball can only reach a column with index j with $r_{init} - n + 1 \leq j \leq r_{init} + n - 1$. Hence, we may check, when setting the parameters $(m = 2k + 1, n, b_{init}, r_{init})$ of the boardgame that $r_{init} + n - 1 \leq m$. Moreover, if $b_{init} + n - 1 < k$ and $r_{init} - n + 1 > k$, then blue and red balls will never interfer (so we should mention it to the user) (the threashold values above should be checked and corrected).

Summarizing: Hence, the parameters that must be set to define a boardgame are n, m, r_{init} and b_{init} with some particular restrictions that must be checked.

After having defined the main game's parameters, the player must design the boardgame by setting the types of the pieces (and possibly their orientation) for each cell (by default, a cell is **empty**). The cell's types are *empty, stop-cell, cross-cell, bascule-cell (Left or Right), bit-cell (initially Left or Right) or gear-cell (initially Left or Right)* for the cells (i, j) with $i + j$ even. The cell's types are *empty or transmitting-gear* for the cells (i, j) with $i + j$ odd.

2 The building of a boardgame

Let us consider a $n \times m$ boardgame with $m = 2k+1$ being odd and an extra cell with coordinates $(n + 1, k + 1)$. Each cell can be of several (five) types and two of the five types may have some (binary) property (called *orientation*). The main point is that depending on the type (and possibly on the property) of a cell, a ball arriving in this cell (from left or from right) will leave it in a direction (left or right) depending on : the type of the cell, its orientation and where the ball comes from (left or right).

2.1 Main pieces to be placed in the cells, case $i+j$ even

2.1.1 Stop cells

Given (i,j) such that $i+j$ is even, a cell (i,j) (at row i and column j) may be a **stop cell**. In this case, any ball arriving from cell $(i-1,j-1)$ or $(i-1,j+1)$ that is reaching the stop-cell (i,j) will be stopped. This ends the current execution/game.

2.1.2 Bascule (or "rocker") cells

Given (i,j) such that $i+j$ is even, a cell (i,j) (at row i and column j) may be a **bascule cell**. A bascule cell has an *orientation*, either *Right* or $Left$. Any ball reaching (from $(i-1,j-1)$ or $(i-1,j+1)$) a bascule-cell (i,j) with Left-orientation will go to the cell $(i+1,j-1)$. If the bascule-cell has Right-orientation, then the ball goes to the cell $(i+1,j+1)$. Here, we need to ensure that any bascule cell at column 1 is oriented toward the right, and any bascule-cell at column m is oriented toward the left).

2.1.3 Cross cells

Given (i,j) such that $i+j$ is even, a cell (i,j) (at row i and column j) may be a **cross cell**. Any ball reaching from $(i-1,j-1)$ (from the upper left) a cross-cell (i,j) will go to the cell $(i+1,j+1)$ (lower right). Any ball reaching from $(i-1,j+1)$ (upper right) a cross-cell (i,j) will go to the cell $(i+1,j-1)$ (lower left).

Note that if there is no way to come from one side to such a cell, it could be replaced by a bascule-cell. This is for instance the case if the cell $(i-1,j-1)$ or the cell $(i-1,j+1)$ is an empty-cell. Also, if $i=n$ and $j<k-1$ or $j>k+1$, a cross-cell (i,j) may be replaced by a bascule-cell.

2.1.4 Bit cells

Given (i,j) such that $i+j$ is even, a cell (i,j) (at row i and column j) may be a **bit-cell**. At each moment, it must be oriented to the Right or to the Left. Here, we must ensure that $j>1$ and $j<m$.

Any ball reaching (from $(i-1,j-1)$ or $(i-1,j+1)$) a bit-cell (i,j) with Left-orientation will go to the cell $(i+1,j-1)$ and, moreover, the orientation of the bit-cell will be switched from Left to Right. If the bit-cell has Right-orientation, then the ball goes to the cell $(i+1,j+1)$ and the bit-cell switch to Left orientation.

2.1.5 Gear cells

Their specifications are exactly the same as for the bit cells. The difference is that they may be "linked" to other gear-cells via the transmitting-gear cells described below.

2.2 Case $i+j$ odd : transmitting-gear cells

Given (i,j) such that $i+j$ is odd, a cell (i,j) (at row i and column j) may be a **transmitting-gear cell**. Their role is to establish "components" of gear-cells that are "linked" with each other. That is, two gear-cells are linked (or belong to the same component) if there is a path from one of them to the other alternating gear-cells and transmitting-gear cells.

More precisely, a gear-cell (i,j) belongs to the same component of any of the transmitting-gear cells in $(i,j-1),(i,j+1),(i-1,j)$ and $(i+1,j)$. So, two gear-cells adjacent to a same transmitting-gear cell will belong to the same component and this is "recursive".

Initially, all gear-cells belonging to a same component must have the same orientation. Then, each time a gear-cell switch its orientation, all the gear-cells related to it (in the same component) change their orientation similarly (even if no ball cross them). So, all along the execution, all gear-cells of a same component must keep same orientation.

3 The initialization and execution of a game

Actually, to start the execution of a game, some other things must be specified such as the orientation of some pieces (which is a "dynamic" way to parameterize the boardgame) and also the number of blue and red balls as inputs.

The precise configuration of the pieces (together with their initial orientation) in the boardgame is also part of the input (with the restriction that they must satisfy the constraints described above). Finally, the number of red/blue balls must also be specified and which is the color of the first ball to be launched.

So, initially, a ball of prescribed color is launched and falls following the rules described above (depending on the boardgame). Once the ball reaches the row $n+1$ (or the virtual row $n+2$ if the ball reaches the cell $(n+1, k+1)$), if it arrives through a column $\leq k$ (resp., $\geq k+2$), it launches a new Blue (resp., Red) ball (if there are still such balls to be launched) and then, the process it repeated from the new current configuration (with the number of available Blue/red balls decreasing by one accordingly). Otherwise, the process stops.

4 Expected Fonctionnalities

4.1 Visualization

As a first step, we propose a "text-mode" TO BE DONE in Python

- $*$: such cell cannot be used (basically because no ball can ever reach it)
- $\cdot$: empty cell with $i+j$ odd
- $\circ$: empty cell with $i+j$ even
- $\backslash$ (resp., $/$) : bascule-cell oriented to Right (resp., Left)
- X : cross-cell
- L (resp., $\checkmark$) : bit-cell oriented to Right (resp., Left)
- $|*$ (resp., $*|$) : gear-cell oriented to Right (resp., Left)
- O : transmitting-gear cell
- B (resp., R) : blue/red ball, at starting position
- $\rightarrow Y$ (resp., $Y \leftarrow$) : Blue (if $Y = B$) or Red (if $Y = R$) ball coming from Left (resp., Right)

4.2 Build boardgame

Ask for the parameters n, m, r_{init}, b_{init}. Start from an empty board with default forbidden $*$ cells and cells $\cdot$ and $\circ$.

Must allow to add a new pieces: ask i,j, the type of cell and, if required, its orientation. Must check whether constraints are satisfied.

Must alert if there is a risk that of forgetting possible paths. For instance, must alert if $(1, r_{init})$ and $(1, b_{init})$ are not assigned some piece. Moreover, when a piece is added at (i, j), must alert if $(i + 1, j - 1)$ or $(i + 1, j + 1)$ is empty.

May alert that some pieces are not required: for instance a gear-cell alone in its component may be replaced by bit-cell, a cross-cell may sometimes replaced by a bascule-cell (see above).

4.3 Loading pre-designed boardgame

Possibly, start from a partial boardgame.

4.4 Executing one step of the game

Given a configuration and the position of one (unique) ball in it, give the next configuration and next position of the ball.

To visualize an execution, we propose to have a second $n * m$ matrix, interleaving with the boardgame matrix, and that a single non empty cell representing the current position of the ball $\rightarrow Y$ (resp., $Y \leftarrow$). An alternative (that seems at first glance to be better to follow the execution) would be to keep visible all previous position (and direction) of the current ball.

4.5 Backtracking one step of the game

Given a configuration and the position of one (unique) ball in it, give the previous configuration and previous position of the ball. This requires to save the previous configurations (actually, since everything is deterministic, the only think that may be recorded where a ball falls (and from where) at the last row... to be verified)

4.6 Evaluating the output of the game

Here, the output may be: where the last ball stops, what is the final configuration after all balls have been launched, what is the order of the released balls... This clearly depends on the problem, but may be factorized into one output (??). Also, the output must be checked for every initial configuration (whatever be initial orientation of the variable pieces). Maybe a "truth-table" would be a good summary of the results?

Previous paragraph would like to evaluate the fact that a given boardgame actually answer the given problem. Another way to evaluate it (if it is correct) could be through a measure describing the "quality" of a boardgame. At a first glance, it could be a lexicographical order on the total number of pieces, the number of "expensive" pieces (gear-cell, bit-cell, cross-cell...).

4.7 Expressing/Visualizing the values of the "registers" in a decimal basis

This requires the definition of registers (part of the output in some boardgames) from a configuration which is not clear yet. Maybe it can be defined explicitly when defining the boargame.

4.8 Play the Turing Tumble game

Propose and evaluate solutions for the Turing Tumble challenges (see next Section(s)). It would be good to better explain each question and the impact/difficulties/performance of the proposed solutions. In particular, I mean that depending on the question, the difficulties may come from the formalisation of the question, the amount of pieces that can be used, the position of the pieces already placed...

5 The 60 Challenges of Turing Tumble

Here, we list the 60 challenges in the game, insisting on their meaning in terms of computer architecture (sometimes considering the "prise en main" of the pieces of the game) and/or algorithmic meaning. Note that some challenges differ in (or/and are not trivial because of) the fact that **they impose a pre-definite partial boardgame**. Here, we do not precisely specify each challenge but rather try to express the main concepts they aim at presenting.

1-2 Understanding **bascule-cells** and how to release a next ball of same color as previously.

3-4 Understanding **bascule-cells** and how to release a next ball of different color: e.g., start with a blue ball and then release all red balls.

5-6 Understanding **cross-cells**: alternate balls of different colors.

7 Understanding **cross-cells**: create path for unique color.

8-10 Understanding **bit-cells**: alternate balls of different colors. **Create simple patterns**: two blue, one red... or: two blue, two red...

11 Write/Impact final state. Bit-cells may carry information. How to modify it? **Flip predefined bits**. Note that it is the first challenge with **input being encoded into the final configuration**, not in the order of the released balls.

12-13 Understanding **stop-cells**: intercept a ball of predefined color.

14-15 Read/Dependance on initial state: depending on the orientation of a bit-cell, intercept a ball with some color.

16-17,21 Counter: intercept the 4^{th} ball. Release 3 blue balls, then 3 red balls and intercept the next ball. Level 21 explain how a **register** (set of bit-cells) encodes the **binary** representation of an integer: Count the number of falling balls.

18-19,36 AND. Intercept a blue ball in left stop-cell if initially two bit-cells point to Left and intercept in a right stop-cell otherwise. Intercept a blue ball if initially two bit-cells point to Left and intercept a red ball otherwise. Level 36 is similar but with a different starting position.

20 OR. Intercept a blue ball if at least one of the two bit-cells starts on the Right, intercept a red ball otherwise.

22-26 Depletion/Decrement. From an initial value encoded by a register (set of bit-cells), decrease it by one for each falling ball. Intercept the i^{th} ball where i is initially encoded in a register. Generate a given pattern of the form x blue balls, then y red balls and then stop. Generate a given pattern of the form 2^t blue balls, then 1 red ball and then 2^t blue balls and then stop (t being fixed, i.e., given as input).

27, 46 Reverse. Reverse the direction of all the 9 bits. Level 46: reverse the direction of the 5 bits all on the same row.

28-29 Understanding **gear-cells**: Release only blue balls. Release one blue, then one red, then only blue balls.

30 Overflow. Count balls, if it goes over some value, change (forever) the orientation of a gear-cell.

31 Write the number of balls in one register amongst two (the one being given by the initial orientation of a gear-cell).

32 Orient a gear-cell depending on which ball (blue or red) is released first.

33 Copy. Given two gear-cells A and B, finally, both must get the same orientation as the initial orientation of A (whatever be the initial orientation of B).

34,43 XOR. Intercept a blue ball if the bit-cell and the gear-cell have distinct initial orientation, intercept a red ball otherwise. Level 43 is the same with 2 gear-cells.

35 Parity. Intercept a blue ball if a register is initially even and a red ball otherwise.

37, 47 SUM of registers. Add the value of register A (3 bits) to the value of register B (4 bits) and store the result in B. If $A + B$ is too big intercept a red ball, and a blue ball otherwise. Same in level 47 (without the interception part) but A has two bits, B has 3 bits and only two red balls may be released.

38 Given 3 bits, return a sequence of four balls such that the $(i+1)^{th}$ ball is red if the i^{th} bit points to Left, and blue otherwise ($i \in \{1, 2, 3\}$). The first ball is blue.

39 Return a sequence: three blue balls, one red, three blue, one red...

40 Multiplication of the number of balls by power of 2. Multiply the number of blue balls by 4 and store the result in a register with 5 bits.

41 Reset. Given a gear-register with 4 bits, set it to zero whatever be its starting position, with one ball.

42, 45, 60 Comparaison of two registers. If $A > B$ (two registers with 4 bits each), intercept a blue ball. Intercept a red ball otherwise (only one stop-cell). Level 45: If $A \leq B$ (each with 3 bits) intercept a blue ball and intercept a red ball otherwise (one stop-cell). In Level 60, there are 3 stop-cells (left, middle and right): intercept a ball in the left stop-cell if $A > B$, in the middle stop-cell if $A = B$ and in the left stop-cell otherwise.

44 Substraction of registers. Compute $B - A$ (assume that $A \leq B$) and store the result in B. Register A has 3 bits and B has 4 bits.

48, 52-53, 55-56 Multiplication/Division of a register. Multiply register A (3 bits, $A \leq 5$) by three and store the result in B (4 bits). In Level 52, A has only 2 bits and must multiplied by 5. In Level 53, a single register A, with 3 gear-bits, must become $A * 2^p$ where p is the number of blue balls released (If $A * 2^p > 7$, we do not care about the final value of the upper bit). Level 55: divide register A (four bits) by 3 and store the result in B (3 bits). In Level 56, Register A consists of 2 bascule-cells (not 2 bit-cells) and so A is **permanent**: its value **cannot be modified**: Multiply register A by B (a register with 2 bit-cells) and store the result in register C (3 bits).

49-51 Infinite Loop. Start from a pattern (8 blue balls on one side, 8 red balls on the other side). These 3 challenges show how to go back to the same pattern (by passing through 2 intermediate patterns). Repeating these programmes will then never stop.

54, 57 Binary to Decimal and vice-versa. Given a register A (2 bits) and 4 bits in the last row, turn to left the i^{th} bit (the bits of the last row are numbered from 0 to 3, from right to left) where i is the number encoded in A (only 2 red balls can be released). Level 57: Given a **permanent** register A (3 bascule-cells in a row such that at most one points to the right), encode i (where $i = 0$ if all bascule-cells point to the left or i is the unique bascule pointing to the right) in a register with 2 gear-bits.

58 Generate the pattern: 2 blue balls, one red, 4 blue, one red and 8 blue balls.

59 Nim Game. Play to the Nim Game against the board. At each turn, the current player takes $1, 2$ or 3 balls. The one who gets the last ball looses. I comment here the solution I proposed. The 2 bascule cells must be initialized depending on the initial number of balls. If the gears cells are turned to the right, it is the "board" turn. If it is turned to the left, it is the human player turn. To take a ball, the human player launches a blue ball (with the gear cells turned to the left). When the human player decides that it is the board turn, he turns the gear cells to the right and launches once the blue ball (the board will proceed until it decides to stop in which case, the gear cells have been turned to the left).

In Section **??**, we present possible solutions to some of the challenges (note that **each challenge may admit several solutions**). Bascule-cells are in green, bit-cells in blue, cross-cells in orange, gear-cells in purple and transmitting-gear cells in red.

6 Going further

While very rich, Turing Tumble cannot do everything. For instance, I think that it is not possible to count the total number of blue and red balls since it is not possible to detect the last ball of some color. Physically, it could be done by having different kind of balls (except for their colors) such that balls with different diameters or weights... This would be an easy property to be added in a software version of Turing Tumble.

There are probably many other simple properties that could be added in order to increase significantly the computational power of circuits that can be built. Your turn to work by thinking to some of them!

7 Solutions of some challenges