

`InitBoard` is another generic function that builds the board and returns the corresponding matrix as a list of lists. It is based on `InitPosition`, whose definition is unique and characteristic of each board game.

```
InitBoard[] := Table[InitPosition[i, j], {i, Height}, {j, Width}]
```

In the example of the chess game, the initial positions are as follows.

```
InitPosition[1, c_] := {2, 3, 4, 5, 6, 4, 3, 2}[[c]]
InitPosition[2, c_] := 7
InitPosition[7, c_] := 13
InitPosition[8, c_] := InitPosition[1, 9 - c] + 6
InitPosition[1_, c_] := 1
```

This produces the usual representation of the chess board at the beginning of the game.

```
InitBoard[] /. x_Integer -> Patterns[[x]] // MatrixForm
```

$$\begin{pmatrix} \{B, R\} & \{B, Kn\} & \{B, B\} & \{B, Q\} & \{B, Kg\} & \{B, B\} & \{B, Kn\} & \{B, R\} \\ \{B, P\} & \{B, P\} & \{B, P\} & \{B, P\} & \{B, P\} & \{B, P\} & \{B, P\} & \{B, P\} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} & \text{empty} \\ \{W, P\} & \{W, P\} & \{W, P\} & \{W, P\} & \{W, P\} & \{W, P\} & \{W, P\} & \{W, P\} \\ \{W, R\} & \{W, Kn\} & \{W, B\} & \{W, Kg\} & \{W, Q\} & \{W, B\} & \{W, Kn\} & \{W, R\} \end{pmatrix}$$

□ Implementing the Transition Function

For most of the board games, playing is just doing one of these three kinds of actions:

- removing a piece from the board
- adding a piece to the board
- moving a piece

So, from an implementation point of view, playing amounts simply to changing the status of one (or two) position(s) of the board and can be completely defined by giving the corresponding location(s) on the matrix as arguments to the transition function. A few games, such as backgammon, may let the player move several pieces during the same play. These cases require further refinement of the function.

Of course, only some positions are playable and, for a selected piece, only some motions are valid with respect to the rules of the game. Those constraints require us to check the arguments of the transition function before its application.

Hence, the generic transition function `PlayThere` relies on two nongeneric functions: `IsPlayable`, which performs the checking, and `PlayBoard`, which changes the board.

```

PlayThere[p : {__Integer}] :=
  Module[{m = IsPlayable[p]}, If[Not[FalseQ[m]], PlayBoard[p, m]]]
PlayThere[{p1_List, p2_List}] := Module[{m = IsPlayable[p1, p2]},
  If[Not[FalseQ[m]], PlayBoard[p1, p2, m]]]

```

FalseQ is analogous to the standard **TrueQ** function and returns **True** only when its argument is **False**.

```

FalseQ[p_] := TrueQ[Not[p]]

```

Many algorithms and computations can be invoked by **IsPlayable** depending on the complexity of the rules of the games. Usually, however, this computation will also provide information on how the board will change due to the play. For example, in chess, if you plan a play to capture a piece, checking this plan will produce the capture as a side effect. So, the results computed by **IsPlayable** usually affect **PlayBoard**. In the implementation, the value returned by the first function is given as an argument to the second.

□ Main Loop and Interaction with the Player

The main loop defines the game as a sequence of plays. It may be used for an interactive game and for batch games for analysis purposes. The generic function **PlayGame** implements this simple loop. We will show how to implement interaction through a notebook graphical interface.

Interaction with the player is made through **View**—for visualization—and through **GetPlay**. **GetPlay** may be used to apply a strategy in a batch game or to prompt the user for the selected play in an interactive game.

```

PlayGame[x___] := (NewGame[x];
  While[Not[GameOver[]], (View[]; PlayThere[GetPlay[]]); EndGame[])

```

The termination phase of the game is performed by **EndGame**, which can be refined for a particular game but has this simple default value.

```

EndGame[] := View[]

```

□ Notebook Visualization

A *Mathematica* notebook, or a palette, can be used for a first approach to graphical visualization and a graphical user interface for a game. Here, we represent the matrix of the board with an array of buttons with text (or a color) for the pattern corresponding to the location. Each of the buttons invokes a transition function—defined in terms of **PlayThere**—with its location as the argument. Control of the game is made directly through the calls and no loop is necessary.

The generic function for playing a new game is **NBPlayGame**, which is analogous to **PlayGame**.

```

NBPlayGame[x___] := (NewGame[x]; Nbview = NotebookPut[NBView[]];)

```

NBPlayGame displays the notebook created by **NBView**.

```

NBView[] :=
  Notebook[{Cell[BoxData[NBBoard[Board]], CellTags → {"board"}]},
  Apply[Sequence, NBOptions]]

```

Inside NBView, NBBoard generically creates the array of buttons.

```

NBBoard[l_] := GridBox[MapIndexed[NBMakeButton, l, {2}],
  RowSpacings → 0, ColumnSpacings → 0]

```

The only nongeneric function to develop when implementing a game is NBMakeButton. It has to take in account the transition of the board at each play, manage the end of the game, and refresh the display. For those purposes, NBMakeButton can use the generic NBPlayThere and NBRefresh. See the following examples for details of implementation.

```

NBPlayThere[p_] := (PlayThere[p]; NBRefresh[]);
NBRefresh[] := (NotebookPut[NBView[], Nbview]);

```

■ The HMaki Example

“If you start this game, you see a board full of tiles. Your task is to remove as many tiles as possible from the board. You cannot remove single tiles, instead you have to remove them in groups of adjacent tiles filled with the same color.” Holger Klawitter, *HMaki 3.9.1 Information* (www.klawitter.de/palm/hmaki.html)



Figure 5. The board of SameGame, the Gnome/KDE version of HMaki.

Let us try our design pattern for implementing this board game (Figure 5).

□ Configuration of the Board

We give some values for the size of the board and the number of colors. We also set the flag for randomness.

```

Width = 12;
Height = 8;
Patterns = Range[5];
NeedRandomness = True;

```

Positions of colored tiles are given by a succession of calls to the pseudorandom number generator.

```

Clear[InitPosition];
InitPosition[l_, c_] := Random[Integer, Length[Patterns] - 1] + 1
Clear[InitPlay]; InitPlay[] := Null

```

□ Visualizing the Board

We are now able to initialize the game.

```

NewGame["new"]

Board // MatrixForm

```

$$\begin{pmatrix} 1 & 4 & 5 & 2 & 2 & 5 & 3 & 4 & 1 & 3 & 3 & 1 \\ 3 & 1 & 5 & 1 & 3 & 5 & 4 & 3 & 3 & 2 & 3 & 3 \\ 1 & 4 & 3 & 1 & 5 & 5 & 4 & 4 & 2 & 4 & 4 & 2 \\ 4 & 2 & 3 & 1 & 4 & 2 & 2 & 2 & 1 & 3 & 1 & 4 \\ 4 & 5 & 2 & 1 & 3 & 4 & 1 & 1 & 1 & 5 & 3 & 1 \\ 3 & 5 & 4 & 4 & 4 & 3 & 5 & 5 & 3 & 2 & 2 & 3 \\ 3 & 5 & 2 & 2 & 1 & 3 & 1 & 4 & 2 & 2 & 2 & 2 \\ 2 & 2 & 3 & 2 & 1 & 4 & 4 & 3 & 5 & 4 & 4 & 4 \end{pmatrix}$$

Choosing some colors, we can define the function View for a nicer display of the board.

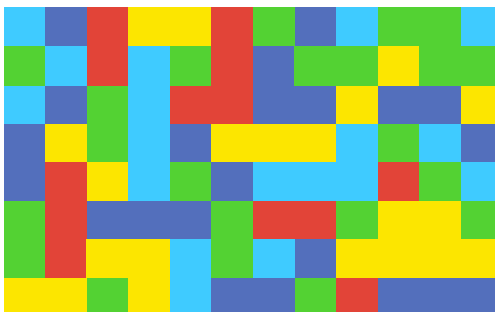
```

Needs["Graphics`Colors`"];
HMakiColors = {LightBeige, DeepSkyBlue,
  CadmiumLemon, LimeGreen, Cobalt, DeepMadderLake};

View[] := Show[
  Graphics[RasterArray[Transpose[Map[Reverse, Transpose[Board]]]]] /.
    x_Integer -> HMakiColors[[x + 1]]]

View[];

```



□ Transition Function

To deal with corner and boundary situations, we define the `BoardValue` function to access the values of the board. It returns -1 if the arguments for location are outside the bounds of the board. This allows us to ignore the boundaries of the board when considering the neighbours of a location.

```
BoardValue[{l_, c_}] :=
  If[Or[Not[0 < l ≤ Height], Not[0 < c ≤ Width]], -1, Board[[l, c]]]
```

The positions of neighbours are computed by the following two functions.

```
FaceNeighbours[p_] :=
  Map[Plus[p, #] &, {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}]
CornerNeighbours[p_] :=
  Map[Plus[p, #] &, {{-1, 1}, {1, 1}, {1, -1}, {-1, -1}}]
```

To get the *spot* enclosing the selected location, (i.e., the neighbourhood of locations with the same color), we use a fixed-point algorithm applied to a neighbourhood extension function.

```
Spot[p_] := FixedPoint[Function[y,
  Union[y, Apply[Union, Map[Select[FaceNeighbours[#], Function[x,
    BoardValue[x] == BoardValue[First[y]]] &, y]]], {p}]
```

`IsPlayable` is now easy to implement: only a non-empty location within a spot of at least two can be played. The result of the function is the list of the locations within the spot.

```
IsPlayable[p_] := If[BoardValue[p] == 0, False,
  Module[{s = Spot[p]}, If[Length[s] > 1, s, False]]]
```

`PlayBoard` will “remove” the spot from the board, propagating individual tiles to the bottom and columns to the left whenever it is possible.

```
PlayBoard[p_, sp_] := Board = Transpose[
  PadRight[Select[Map[PadLeft[Select[#, Positive], Height] &,
    Transpose[ReplacePart[Board, 0, sp]]], Positive[
    Apply[Plus, #] &], Width, x] /. x → Table[0, {i, Height}]]];
```

□ Main Loop

To check whether the game is over, it is necessary to decide if there remain two spots of the same color contiguous by a face.

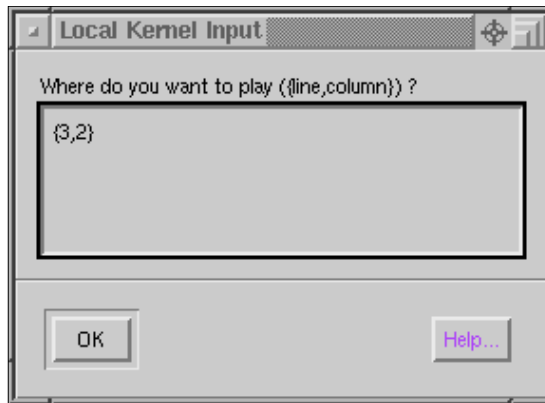
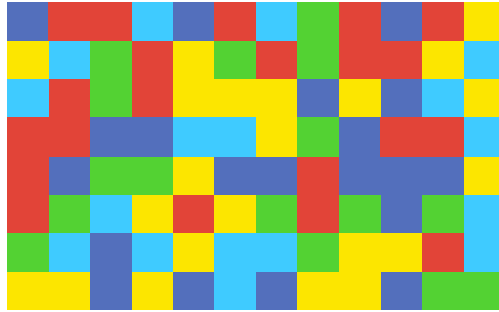
```
GameOver[] := And[Length[Select[Select[Flatten[Map[Split, Board], 1],
  Length[#] != 1 &], First[#] != 0 &]] == 0,
  Length[Select[Select[Flatten[Map[Split, Transpose[Board]], 1],
  Length[#] != 1 &], First[#] != 0 &]] == 0]
```

Interaction with the player here is very basic.

```
GetPlay[] := Input["Where do you want to play ({row,column}) ? "]
```

And we are able to play—but the interface is a bit cumbersome.

`PlayGame["new"]`



With a Notebook Interface

To improve the interface, only one function has to be defined: one which colors the buttons and manages the end of the game (Figure 6).

```
NBMakeButton[c_, {i_, j_}] := ButtonBox[" ", ButtonData -> {i, j},
  Background -> HMakiColors[[c + 1]], ButtonFunction ->
  (If[Not[GameOver[]], NBPlayThere[#2] &, NotebookClose[Nbview]]),
  ButtonEvaluator -> Automatic]
```

And we are able to play without the keyboard!

```
NBPlayGame["new"]
```

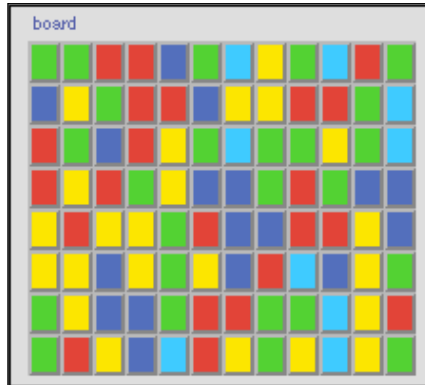


Figure 6. A view of the notebook interface of the board of HMaki.

■ Playing Lines

“Your task is to build lines of balls of the same color on the checkerboard. Every time you move a ball, 3 new balls appear. When you build a line of 5 or more balls, these balls are removed from the board. Easy? And exciting!!!” *5star Free Lines—How to play* (www.atomax.com/lines.stm)

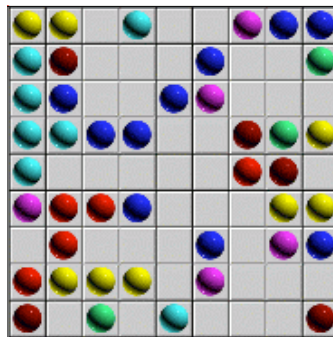


Figure 7. The board of the five-star version of the game Lines.

Once again, the design pattern will help us quickly implement this game (Figure 7).

□ Configuration of the Board

The following values define the physical parameters of the board.

```

Width = 9;
Height = 9;
Patterns = Range[7];
NeedRandomness = True;

```

Initially, the board is empty.

```

Clear[InitPosition]
InitPosition[l_, c_] := 0

```

But before the first play, the computer makes three balls appear.

```

Clear[InitPlay]
InitPlay[] := (Next[]; AddBall[NextBall]; Next[]; FromLoc = {};)

```

The colors of the next three balls to be added are selected in advance and must be shown to the player to help him play. They are selected randomly by the `Next` function and kept in a global variable, `NextBall`.

```

Next[] :=
  (NextBall = Table[RandomElement[Range[Length[Patterns]]], {3}];)

```

Adding Balls

The positions at which to add the balls are also selected randomly in the list of the empty positions.

```

AddBall[l_List] := Map[AddBall, l]
AddBall[c_Integer] := Module[{p = RandomElement[Position[Board, 0]]},
  (Board = ReplacePart[Board, c, p]; NewLine[p];)]

```

`RandomElement` is a miscellaneous function randomly selecting an element of a list.

```

RandomElement[l_List] := Part[l, Random[Integer, Length[l] - 1] + 1]

```

□ Visualizing the Board

Let us initialize the game and visualize it.

```

NewGame["new"]

```

With some colors and a few graphics primitives, we are able to display the board and the next three balls.

```

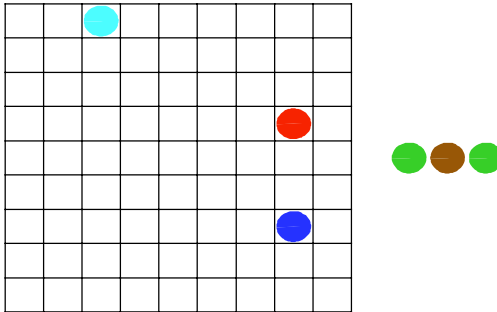
LinesColors = {RGBColor[0.8, 0.8, 0.8], Blue, Red,
  PermanentGreen, Yellow, SaddleBrown, Cyan, Magenta};

ViewBall[{i_, {l_, c_}}] :=
  {LinesColors[[i + 1]], Disk[{c - 0.5, Height - 1 + 0.5}, 0.45]}

View[] := Show[
  Graphics[Join[Map[Line[{{#, 0}, {#, Height}}] &, Range[0, Width]],
    Map[Line[{{0, #}, {Width, #}}] &, Range[0, Height]],
    Map[ViewBall, Module[{p = Position[Board, x_ /; x != 0]},
      Transpose[{Extract[Board, p], p}]]],
    MapIndexed[ViewBall[{{#1, {5, 10 + #2[[1]]}}] &, NextBall]]]]]

```

```
View[];
```



□ Transition Function

In this game, a play consists of three possible successive actions: selecting and adding three balls, moving a ball, and deleting a line of balls. When a player selects a ball and a new location for it, this ball is moved. If possible a line is deleted and the computer adds three balls, possibly deleting a line.

Moving Balls

To decide whether a motion is valid or not, we again use a fixed-point algorithm applied to a neighbourhood extension function: the set of reachable positions is incrementally built from the starting position by exploring empty locations.

We reuse the function `BoardValue` to simplify the computations of the neighbourhood.

```
BoardValue[{l_, c_}] :=
  If[Or[Not[0 < l ≤ Height], Not[0 < c ≤ Width]], -1, Board[[l, c]]]
```

`Neighbour` is slightly different than `FaceNeighbour` from the previous example.

```
Neighbour[{l_, c_}] :=
  {{l - 1, c}, {l, c - 1}, {l, c}, {l, c + 1}, {l + 1, c}}
```

Possible searches for empty locations in the neighbourhood.

```
Possible[{l_Integer, c_Integer}] :=
  Select[Neighbour[{l, c}], MemberQ[Position[Board, 0], #] &]
Possible[lp_List] := Apply[Union, Map[Possible, lp]]
```

The fixed-point algorithm is performed by `IsPlayable`, which returns a Boolean.

```
IsPlayable[p1_, p2_] := And[BoardValue[p1] != 0,
  BoardValue[p2] == 0, MemberQ[FixedPoint[Possible, p1], p2]]
```

`PlayBoard` updates the board, checks for lines of five balls to delete, and adds three balls.

```

PlayBoard[p1_, p2_, _] :=
  (Board = ReplacePart[Board, BoardValue[p1], p2];
   Board = ReplacePart[Board, 0, p1];
   If[Not[NewLine[p2]], (AddBall[NextBall]; Next[]);])

```

Checking for Lines to Delete

To check if there is a line of five balls to delete, we try to extend the position in four (nonoriented) directions as long as the color of the ball remains the same. This is performed by `Extend`.

```

Extend[pos_, dir_] :=
  FixedPoint[If[BoardValue[First[#] - dir] == BoardValue[First[#]],
    Prepend[#, First[#] - dir], #] &,
  FixedPoint[If[BoardValue[Last[#] + dir] == BoardValue[Last[#]],
    Append[#, Last[#] + dir], #] &, pos]]
Extend[pos_] := Map[Extend[{pos}, #] &,
  {{1, 0}, {1, 1}, {0, 1}, {-1, 1}}]

```

`DeleteBall` suppresses the balls from the line.

```
DeleteBall[p_] := (Board = ReplacePart[Board, 0, p];)
```

And `NewLine` is the top-level function, returning a Boolean.

```

NewLine[p_] :=
  Module[{l = Flatten[Select[Extend[p], Length[#] > 4 &], 1]},
    (Map[DeleteBall, l]; Not[l == {}])]

```

□ **Main Loop**

Playing

The game continues until fewer than three slots are available for the balls.

```
GameOver[] := Length[Select[Flatten[Board], # == 0 &]] < 3
```

Once again, the interface with the player is extremely basic.

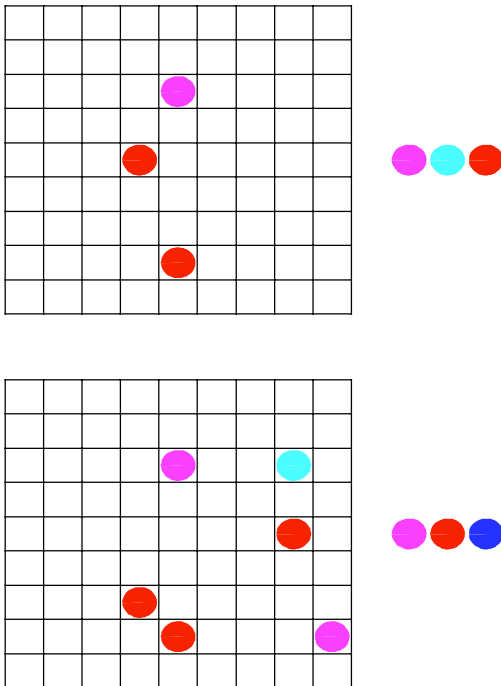
```

GetPlay[] := Input["which motion (
  {start_line,start_column},{end_line,end_column})?"]

```

But we can play!

PlayGame["new"]



□ A Notebook Interface

To cope with the display of the “balls to come”, it is necessary to adapt the NBView function and develop the dedicated function NBNext, which is similar to NBBoard.

```
NBView[] :=
  Notebook[{Cell[BoxData[NBNext[NextBall]], CellTags -> {"next"}],
    Cell[BoxData[NBBoard[Board]], CellTags -> {"board"}]},
  Apply[Sequence, NBOptions]]

NBNext[l_] :=
  GridBox[Map[ButtonBox[" ", Background -> LinesColors[#+1]] &, 1]]
```

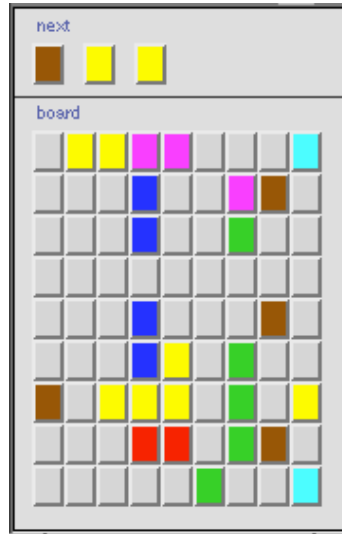
To select two locations, the player clicks twice in a play on the board. This is taken into account through the variable FromLoc and by a small difference in the implementation of NBPlayThere.

```
NBMakeButton[c_, {i_, j_}] :=
  ButtonBox[" ", ButtonData -> {i, j}, Background -> LinesColors[c+1],
  ButtonFunction -> (If[Not[GameOver[]], NBPlayThere[{FromLoc, #2}] &,
  NotebookClose[Nbview]]), ButtonEvaluator -> Automatic]

NBPlayThere[{{}, p_] := (FromLoc = p; NBRefresh[])
NBPlayThere[p_] := (PlayThere[p]; FromLoc = {}; NBRefresh[])
```

With the notebook interface, playing becomes easier, even if the graphical design is not so nice.

```
NBPlayGame["new"]
```



■ Mancala: A Two-Player Game

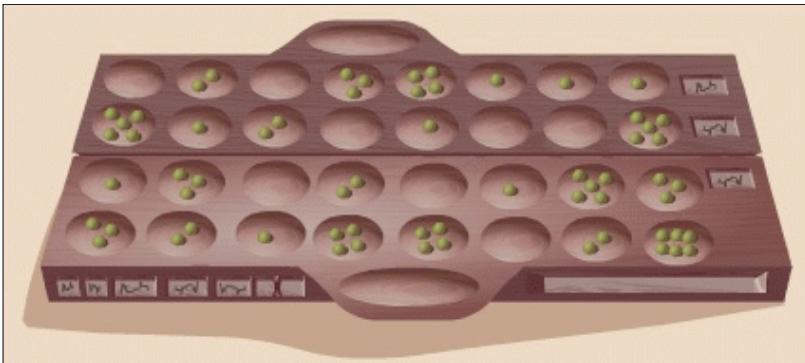


Figure 8. A board of 4 x 8 Mancala.

Mancala is the ancient game of counting and strategy where each player must attempt to collect as many stones as possible before one of the players clears his side of stones (Figure 8). There are many versions of this traditional game.

Here are the rules of the version we will implement:

