Supplementary materials

If you are familiar with Cormas and Smalltalk, you can be interested in more details about the application and how it was developped.

As mentioned in the article, a MainGUI class was defined as a subclass of Cormas, the main window. It was important to work in a subclass, permitting to add or redefine some specific variables and method. The first obvious example of this redefinition is about the #windowSpec method. The GUI is now designed in a unique window containing all the widgets.

Some tricks in the VisualWorks widgets were used :

- Most of the buttons (the icons) have an image as label and their 'Bordered' property is unchecked, making them as 'sensitive icon'
- Some basic views are integrated using the 'View Holder' widget. It is reserved for specific graphical views like the one appearing in the phase when players draw freely their walk preferred track
- Finally, the space interface, by default opened in a second window by Cormas, is integrated using the VisualWorks concept called 'SubCanvas'

This last integration caused more difficulties. The SpaceInterface, which encapsulates the SpaceView, is designed to be a main window and it should be refined in order to redesign its appearance and to cut its menu capabilities. Its content, the SpaceView, has no specificities and is used as is, but a big work was done with its model, the SpaceModel and its controller. Finally, these classes were subclassed using an 'Integrable' suffix, making obvious their specific behaviour in an integration in the MainGUI.

A method was redefined in the SpaceControllerIntegrable class, removing the menu:

processMenuAt: aPoint centered: centered

"we don't open any menu"

Some others were redefined in the SpaceInterfaceIntegrable class for the same purpose :

initMenuAttributs

"we have no menu to init"

updateName

"we no longer change the main window title"

Many others methods were redefined in the SpaceModelIntegrable class in order to inhibit some standard behavior. For example :

performCtrlRedButton: aPoint

"no CtrlRedButton action desired"

Finally, the assembly of these elements is done in the early called methods #initialize, #endLoadModel and #prepareVisualizationSpace in the class MainGUI. Basically, some extracting was done from the original Cormas's methods since its classes were subclasses and the model is now considered already loaded in the system.

initialize

"Init the game and its components.

We are taking some piece of init methods since we aggregate all in a unique window not yet opened. We connect the spaceModel with the main window making accessible the display of the currentMode on the graphicsContext (see the method #displayMode:)"

```
super initialize.
TimeZone        "initialize the timezone for France (MottesPiquet is not internationalized)"
   setDefaultTimeZone:
      (TimeZone timeDifference: 1 DST: 1 start: 2 end: 3 from: 90 to: 304
         startDay: #Sunday).    "on March 31"    "until October 31"
self endLoadModel. "<-load the colours too"
CormasNS.Models.MottesPiquet.Joueur colorsDict
   keysAndValuesDo: [:k :v |
      self class colors at: k put: v]. "copy the colours loaded from the file"
self class createIcons. "create all icons (some are based on the colours loaded from the ev file)"
self prepareVisualizationSpace.   "init the visualization (SpaceModel, SpaceView, ...)"
self currentSimulation spaceModel mainGUI: self. "connect the spaceModel with the mainGUI"
self log: 'Cliquez sur le bouton "Initialisation" pour commencer' asText allBold. "log the first explain text in the log view"
self photos selectionIndexHolder onChangeSend:  #selectionBilanChanged to: self. "prepare the dependency for the snapshots review"
gameOver := false.    "a boolean indicating the end of the game (permit the review of the snapshots)"
```

endLoadModel

```
"extract from #loadModel, copy of the initialization part.
We suppose an already and correctly loaded model, the 'convertToVW73' is no longer necessary"

self currentModel: CormasNS.Models.MottesPiquet.MottesPiquet.  "set the current model class"
self currentModel initialize. "init the model"
self loadEV_file.            "load the EV file (it's used for colors and other parameters configuration)"
self currentSimulation: self currentModel new.  "créate the instance (the currentSimulation in Cormas)"
self createPdvDict.     "initialize the point of view of the model"
self resetSimulation.    "reset some internal counter of the simulation"
self cycle value: 10.   "MottesPiquet is based on a 10 cycles (each one simulating 2 years)"
```

prepareVisualizationSpace

```
"prepare the visualization space.
The Model-View-Controller paradigme is not very well respected in Cormas, we have to copy some pieces of the #openView:"

| legend |
self currentSimulation spaceModel initializeView.  "the model initialize the SpaceView"
self viewWin: CormasNS.Kernel.SpaceInterfaceIntegrable new.  "créate the SubCanvas we integrate in the mainGUI
```

```
    self viewWin model: self currentSimulation spaceModel.  "connect the spaceModel"
    legend := NullLegend new.    "a NullLegend inhibits the awful method #majLegendes:)"
    self viewWin legend: legend.  "replace the legend since we don't want any one"
    self viewWin vue mainInterface: self viewWin. "connect the spaceView and its containing
spaceInterface "
    self currentSimulation spaceModel createSpatialEntitiesImages. "create the graphical
elements"
```

The main method (#validate) in the system is called with the button "Valider". It mainly call the #runStepByStep after verifying different rules of the game or setting some variables helping to verify these rules at the next step.

Finally, the most important method is #displayMode: in the MainGUI class :

displayMode: anInteractionMode

```
    "display anInteractioMode. We display the icon associated with this mode until a mouse
clic."

    | aSensor |
    self updateGUI.        "recompute and display the states of all widgets"
    anInteractionMode action = #selection ifTrue: [^self].  "the default interaction mode has no
special display"
    aSensor := self mainWindow controller sensor.       "access to the sensor (the mouse
device)"
    aSensor waitNoButton.              "wait for no pressed button. It's important since we will
display until a button pressed"
    Cursor blank                       "display a blank cursor"
        showWhile: [
            | icon offset |
            icon := anInteractionMode icon.      "get the icon from the mode (it can be a
MultiIcon)"
            offset := anInteractionMode offset.   "get its offset (usefull for correctly defining the
'clicking' pixel in the icon)"
            icon
                follow: [aSensor cursorPoint - offset]    "the icon follow the mouse pointer"
                while: [aSensor redButtonPressed not]     "until we clic with the 'red' button (usually
the 'left' button)"
                on: self mainWindow graphicsContext].     "on the MainGUI's graphicsContext"
```

You can notice this method doesn't handle the click. It's the responsibility of each widget to do that.

Now we can get a better understanding of how this works with some details about two concrete subclasses. The InteractionModePawn is responsible for carrying one or more pawns.
Its redefined methods are:

action

```
    "the mode pawn is identified par the action #pawn"

    ^#pawn
```

pawn
"return the carried pawn if any one is effectively carried"

self pawns isEmpty ifTrue:[^nil].
^self pawn first

pawns
"return all carried pawns"

^pawns ifNil: [pawns := OrderedCollection new]

pawns: somePawns
"set the carried pawns"

pawns := somePawns

pawn: aPawn
"set the carried pawn"

self pawns: (OrderedCollection with: aPawn)

icon
"return the icon to be displayed when this mode is active"

self pawns isEmpty ifTrue: [^MainGUI unkownIcon]. "should not occurs"
self pawns size = 1 ifTrue: [^self pawn icon]. "take the icon from the pawn"
^MainGUI iconWithPrefix: #multipleIcon name: self pawn type "take a prebuilt multiple icon for this pawn type"

Two others methods will be explained later.

An instance of this class is initiated by clicking on buttons like sheep, cow, rabbit or tractor, which calls a generic method activateModePawn: aTypeSymbol

activateModePawn: aTypeSymbol
"activate a pawn mode for the type aTypeSymbol"

| leftPawns pawn currentMode |
currentMode := self interactionMode. "get the current mode"
(currentMode action = #pawn and: [currentMode pawn type = aTypeSymbol]) "if the mode is already a pawn mode for this type"
ifTrue: [^self interactionMode: InteractionModeSelection new]. "then we have clicked again on the current mode. The new mode should be the selection mode"
leftPawns := OrderedCollection new. "get the actually left pawns"
leftPawns addAll: (self freePawnsOfType: aTypeSymbol). "all pawns are kept by the model"
leftPawns isEmpty "some pawns can be secretly created by clicking on the gray icon"
ifTrue:

```
        [pawn := self createIfPossibleAPawnOfType: aTypeSymbol.          "try to create a
new pawn (they are stricts rules when you can do that"
        pawn isNil
            ifTrue:                                          "if no new pawn raise a dialog"
                [Dialog
                    warn: 'All pawns of type ' , aTypeSymbol , ' are already placed !'.
                ^self interactionMode: currentMode]                  "Then the current mode
remains"
            ifFalse: [leftPawns add: pawn]].                        "if a new pawn is created, it will
carried with the new mode"
    self interactionMode: (InteractionModePawn withPawns: leftPawns)       "create the new
mode carrying the pawns"
```

This last message call the #displayMode: seen previously.

The last interesting method is called by the SpaceModelIntegrable when its controller detect a click:

performRedButton: aPoint

```
    "The user has clicked at aPoint. We search the corresponding square and let the current
mode do its job"

    | square |
    (square := self detectSpatialEntityImageIncluding: aPoint) isNil
        ifTrue: [^nil].
    self interactionMode interactWith: self in: square at: aPoint
```

The last method we have to see is that #interactWith:in:at: in the InteractionModePawn.

interactWith: aSpaceModel in: aSquare at: aPoint

```
    "interact with the square doing what the mode has to do.
    If we carry a pawn and the rules are respected we drop it in the square"

    | nextMode |
    nextMode := nil.                              "prepare the next mode after this interaction"
    self pawn                                    "if we are carrying a pawn"
        ifNotNil:
            [:p |
            (self isAcceptable: p forSquare: aSquare)    "is that pawn can be dropped in the square
?"
                ifTrue:
                    [p moveTo: aSquare centre.          "move the pawn is the square (aSquare is a
group of 16 cells with one called centre)"
                    self pawns remove: p.              "remove the pawns from the carried since it is
dropped"
                    self pawns isEmpty ifFalse: [nextMode := self]]     "the current mode remains
current if we have more pawns to carry"
                ifFalse: [nextMode := self]].                    "the current mode remains current
even if the drop is not acceptable"
    nextMode ifNil: [nextMode := InteractionModeSelection new].        "nextMode is nil if the
last pawn is dropped (or if no pawn was carried"
```

unSpaceModel interactionMode: nextMode "reset the current mode for the next loop"

Four others InteractionMode are built with the same approach :

- InteractionModeEtrepage : for the tool <etrepage>
- InteractionModeHouse : for disposing the houses of the players
- InteractionModeTrack : for the players draw their preferred walk track
- And finally the default InteractionModeSelection which display nothing but can pickup any pawn and launch action from the other buttons