Building a memory game with Bloc

Andrei Chiș, Stéphane Ducasse and Aliaksei Syrel

May 28, 2017

master @ f7d2fba*

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Objectives of this book

Bloc's design is getting stable and this book is a first tutorial on Bloc. Some elements may change such as the name of certain methods, but most of these changes will be minor.

In this tutorial you will build a memory game. We provide the model and focus on creating the UI of the game.

1.1 Memory game

Let us have a look at what we want to build with you: a simple Memory game. In a memory game players need to find pairs of similar cards. In each round a player turns over two cards at a time. If the two cards show the same symbol they are removed and the player gets a point. If not, they are both flipped.

For example, Figure 1-1 shows the game after the first selection of two cards. Face-down cards are represented with a cross and turned cards are just showing a number. Figure 1-2 show the same game after a few rounds. While this game can be played by multiple playes, in this turorial we will build a game with just one player.

To start with, here is the code that builds and launches the game:

```
game := MgdGameModel new initializeForSymbols: '12345678'.
grid := MgdGameElement new.
grid memoryGame: game.

space := BlSpace new.
space extent: 420@420.
space root addChild: grid.
space show
```

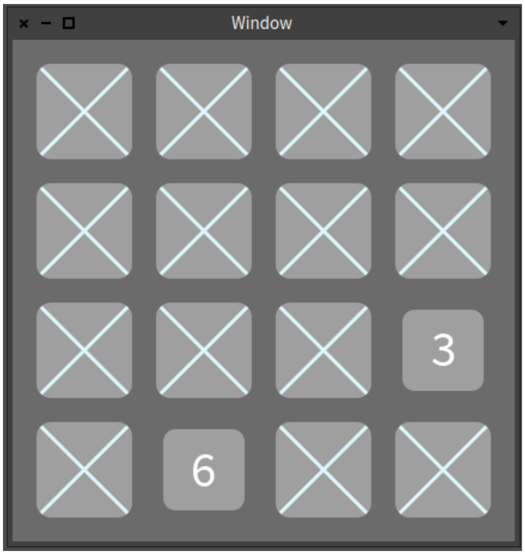


Figure 1-1 The game after the player selected two cards: faced-down cards are represented with a cross and turned card with their number.

- first, we create a grame model and ask to get the numbers from 1 to 8 associated with the cards. By default a game model has a size of 4 by 4, which requires eight different cards.
- Second, we create a graphical game element.
- Third, we assign the model of the game to the UI.
- Finally, we create a graphical space in which we place the game UI and we display the space.

1.2 Getting started

This tutorial is for Pharo 60 running on the latest Pharo60 Virtual machine. You can get them at the following address

```
http://get.pharo.org/60+vm
```

Alternatively, you can download them by executing the line below on a Linux or MacOs system:

```
wget -O- get.pharo.org/60+vm | bash
```

To load Bloc execute the following snippet:

```
Metacello new
    baseline: 'Bloc';
```

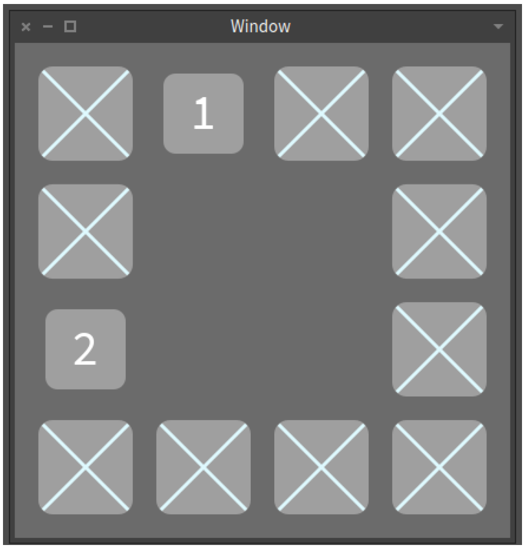


Figure 1-2 Another state of the memory game after the player correctly matched two pairs.

```
    repository: 'github://pharo-graphics/Bloc/src';
    load: #core.
```

1.3 Loading the Memory Game

To make the demo easier to follow and help you if you get lost we already made a full implementation of the game. You can load it using the following code:

```
Metacello new
    baseline: 'BlocTutorials';
    repository: 'github://pharo-graphics/Tutorials/src';
    load
```

After you loaded the BlocTutorials project, you will get two new packages: `Bloc-MemoryGame` and `Bloc-MemoryGame-Demo`. `Bloc-MemoryGame` contains the full implementation of the game. Just to the class side of `MgExamples` and click on the gree triangle next to the `open`method to start the game. `Bloc-MemoryGame-Demo` is a skeleton for the game that we will use in this tutorial.

CHAPTER 2

Game model insights

Before starting with the actual graphical elements, we first need a model for our game. This game model will be used as a model in the typical Model View architecture. One the one hand, the model does not communicate directly with the graphical elements; all communication is done via announcements. On the other hand, the graphic elements are communicating directly with the model.

In the remainder of this chapter we describe the game model in details. If you want to move directly to building graphical elements using Block the package `Bloc-MemoryGame-Demo` already contains the model.

2.1 Reviewing the card model

Let us start with the card model: a card is an object holding a symbol to be displayed, a state representing whether it is flipped or not and an announcer to emit state changes. This object could also be a subclass of Model which already provide announcer management.

```
Object subclass: #MgdCardModel
  instanceVariableNames: 'symbol flipped announcer'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Demo-Model'
```

After creating the class we add an `initialize` method to set the card as not flipped, together with several accessors:

```
MgdCardModel >> initialize
  super initialize.
  flipped := false
```

```
MgdCardModel >> symbol: aCharacter
  symbol := aCharacter
```

```
MgdCardModel >> symbol
  ^ symbol
```

```
MgdCardModel >> isFlipped
  ^ flipped
```

```
MgdCardModel >> announcer
  ^ announcer ifNil: [ announcer := Announcer new ]
```

Next we need two API methods to flip a card and make it dissaper when it is no longer needed in the game.

```
MgdCardModel >> flip
  flipped := flipped not.
  self notifyFlipped
```

```
MgdCardModel >> disappear
  self notifyDisappear
```

The notification is implementing as follows in the `notifyFlipped` and `notifyDisappear` methods. They simply announce events of type `MgdCardFlippedAnnouncement` and `MgdCardDisappearAnnouncement`. The graphical elements will have to register subscriptions for these announcements.

```
MgdCardModel >> notifyFlipped
  self announcer announce: MgdCardFlippedAnnouncement new
```

```
MgdCardModel >> notifyDisappear
  "Notify all observers that I disappeared from the game"
  self announcer announce: MgCardDisappearAnnouncement new
```

Here, `MgdCardFlippedAnnouncement` and `MgCardDisappearAnnouncement` are just subclasses of `Announcement`.

```
Announcement subclass: #MgdCardFlippedAnnouncement
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Demo-Events'
```

```
Announcement subclass: #MgdCardDisappearAnnouncement
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Demo-Events'
```

We add one final method to print a card in a nicer way and we are done with the card model!

```
MgdCardModel >> printOn: aStream
  aStream
    nextPutAll: 'Card';
    nextPut: Character space;
    nextPut: $(;
    nextPut: self symbol;
    nextPut: $)
```

2.2 Reviewing the game model

The game model is simple: it keeps the tracks off all the available cards and all the cards currently selected by the player.

```
Object subclass: #MgdGameModel
  instanceVariableNames: 'availableCards chosenCards'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Demo-Model'
```

```
MgdGameModel >> initialize
  super initialize.
  availableCards := OrderedCollection new.
  chosenCards := OrderedCollection new
```

```
MgdGameModel >> availableCards
  ^ availableCards
```

```
MgdGameModel >> chosenCards
  ^ chosenCards
```

We hardcode for now the size of the grid and of the number of cards that need to be matched by a player.

```
MgdGameModel >> gridSize
  "Return grid size, total amount of card is gridSize^2"
  ^ 4
```

```
MgdGameModel >> matchesCount
  "How many choosen cards should match in order for them to
    disappear"
  ^ 2
```

```
MgdGameModel >> cardsCount
  "Return how many cards there should be depending on grid size"
  ^ self gridSize * self gridSize
```

To initialize the game with cards we add a dedicated method, `initializeForSymbols:`. This method creates a list of cards from a list of characters and shuffle it. We also add an assertion in this method to verify that the called provided enough characters.

```
MgdGameModel >> initializeForSymbols: characters

  self
    assert: [ characters size = (self cardsCount / self
    matchesCount) ]
    description: [ 'Amount of characters must be equal to possible
    all combinations' ].
  availableCards := (characters asArray collect: [ :aSymbol |
    (1 to: self matchesCount) collect: [ :i |
       MgCardModel new symbol: aSymbol ] ])
          flattened shuffled asOrderedCollection
```

Next we need `chooseCard`, a method that will be called when a user selectes a card. This method is actually the most complex method of the model and implements the main logic of the game. First, the method makes sure that the seleted card is not already selected. This could happen if the views uses animations that give players the change to click on the card more then once. Next, the card is flipped by sending it the message `flip`. Finally, depending on the actual state of the game the step is complete and the selected cards removed, or all selected cards are flipped back.

```
MgdGameModel >> chooseCard: aCard
  (self chosenCards includes: aCard)
    ifTrue: [ ^ self ].
  self chosenCards add: aCard.
  aCard flip.
  self shouldCompleteStep ifTrue: [
    ^ self completeStep ].
  self shouldResetStep ifTrue: [
    self resetStep ]
```

The current step is completed if the player selected the right ammont of cards and they all show the same symbol. In this case, all selected cards receive the message `dissapear` and are removed from the list of selected cards.

```
MgdGameModel >> shouldCompleteStep
  ^ self chosenCards size = self matchesCount and: [
    self chosenCardMatch ]
```

```
MgdGameModel >> chosenCardMatch
  | firstCard |
  firstCard := self chosenCards first.
  ^ self chosenCards allSatisfy: [ :aCard |
    aCard isFlipped and: [ firstCard symbol = aCard symbol ] ]
```

```
MgdGameModel >> completeStep
  self chosenCards
    do: [ :aCard | aCard disappear ];
    removeAll.
```

The current step should be reset if the player selected a third card. This will happen when a player already selected two cards that did not match and clicked on a third one. In this sitution the initial two cards will be flipped back. The list of selecte cards will only contain the third card.

```
MgdGameModel >> shouldResetStep
  ^ self chosenCards size > self matchesCount
```

```
MgdGameModel >> resetStep
  |lastCard|
  lastCard := self chosenCards  last.
  self chosenCards
    allButLastDo: [ :aCard | aCard flip ];
    removeAll;
    add: lastCard
```

2.3 Ready

We are now ready to start building the game view.

Since Bloc is still under development, it may happen that you will get exceptions after which graphical elements do not render correctly. In that case the Universe has to be reinitialized.

```
BlUniverse reset
```

CHAPTER 3

Building card graphical elements

In this chapter we will build step by step the visual appearance of the cards. In Bloc visual objects are called elements and usually you define subclass of `BlElement` the inheritance tree root. In subsequent chapters we will do the same for the game and add interaction in terms of event listeners.

3.1 First: the card element

A graphic element is a subclass of the `BlElement`. It simply has a reference to a card model.

```
BlElement subclass: #MgdCardElement
  instanceVariableNames: 'card'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Tutorial'
```

The message `backgroundColor` is one of the customisation hooks defined in `BlElement`. Let us define a nice color.

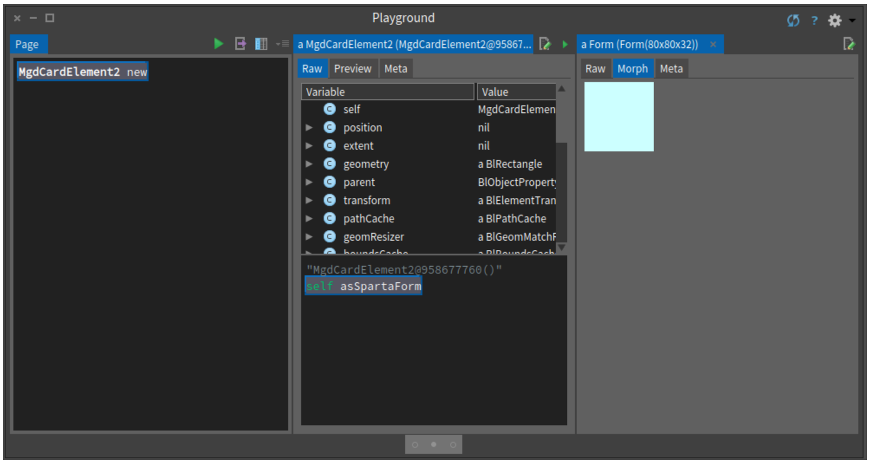
```
MgdCardElement >> backgroundColor
  ^ Color lightBlue
```

We mentioned the accessors since the setter will be place to hook registration for the communication between the model and the view.

```
MgdCardElement >> card
  ^ card
```

```
MgdCardElement >>   card: aMgCard
    card := aMgCard
```

We initialize it to get a



```
MgdCardElement2 new
```

Variable	Value
self	MgdCardElemen
position	nil
extent	nil
geometry	a BlRectangle
parent	BlObjectPropert
transform	a BlElementTran
pathCache	a BlPathCache
geomResizer	a BlGeomMatchF

```
"MgdCardElement2@958677760()"
self asSpartaForm
```

Figure 3-1 A first extremely basic representation of face down card.

```
MgdCardElement >>   initialize
  super initialize.
  self size: 80 @ 80.
  self card: MgdCardModel new
```

3.2 Starting to draw a card

To define the visual properties of a graphic element we redefine the method `drawPathOnSpartaCanvas:`.

(This method will be renamed `drawPathOn:` in the future).

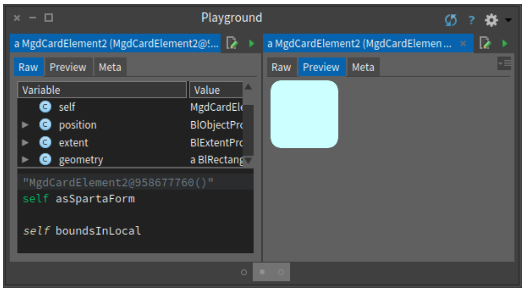
```
MgdCardElement >> drawPathOnSpartaCanvas: aCanvas

  super drawPathOnSpartaCanvas: aCanvas.
  aCanvas fill
    paint: self backgroundColor;
    path: self boundsInLocal;
    draw
```

Note that if we forget to send the message draw the canvas will be set but it will not display the result.

Now to see the result in Morphic we have to get a spartaForm as follows:

```
MgdRawCardElement new asSpartaForm
```

You can also use the the inspector as shown in Figure 3-1. Here we create and inspect the graphic element and then we ask it form and look at it in the Morph pane (this is what the Preview pane is actually doing).


Playground
a MgdCardElement2 (MgdCardElement2@9...
a MgdCardElement2 (MgdCardElemen...
Raw
Preview
Meta
Raw
Preview
Meta

Variable	Value
self	MgdCardEle
position	BlObjectPro
extent	BlExtentPro
geometry	a BlRectang

```
"MgdCardElement2@958677760()"
self asSpartaForm

self boundsInLocal
```

Figure 3-2 A rounded card.

3.3 Improving the card visual

Instead of displaying a full rectangle, we want a better visual. Bloc offers a shape factory. This shape factory returns shape path (lines, rectangle, ellipse, circle...) that can be passed to the canvas using the message `path:`. Other shapes can be easily added.

For example with the following expression `path: (aCanvas shape ellipse: self boundsInLocal)` we draw now a circle since the bounds of the receiver returns a square of 80.

```
MgdCardElement >> drawPathOnSpartaCanvas: aCanvas

  | radius |
  super drawPathOnSpartaCanvas: aCanvas.
  radius := 12.
  aCanvas fill
    paint: self backgroundColor;
    path: (aCanvas shape ellipse: self boundsInLocal);
    draw
```

For our card we would like to have a rounded rectangle so we use the `roundedRectangle:radii:` factory message.

```
MgdCardElement >> drawPathOnSpartaCanvas: aCanvas

  | radius |
  super drawPathOnSpartaCanvas: aCanvas.
  radius := 12.
  aCanvas fill
    paint: self backgroundColor;
    path: (aCanvas shape roundedRectangle: (self boundsInLocal)
    radii: (BlCornerRadii radius: 12));
    draw
```

You should get then a visual aspect close to the one shown in Figure 3-2.

3.4 Preparing flipping

We define now two methods

```
MgdCardElement >> drawBacksideOn: aCanvas
  "nothing for now"
```

```
MgdCardElement >> drawFlippedOnCanvas: aCanvas
  "nothing for now"
```

And we refactor `drawPathOnSpartaCanvas:` as follows: we extract the common part into a separate method.

```
MgdCardElement >> drawCommonOnCanvas: aCanvas
  | radius |
  super drawPathOnSpartaCanvas: aCanvas.
  radius := 12.
  aCanvas fill
    paint: self backgroundColor;
    path: (aCanvas shape roundedRectangle: self boundsInLocal radii:
    (BlCornerRadii radius: 12));
    draw.
```

Finally, `drawPathOnSpartaCanvas:` logic is at the same conceptual level.

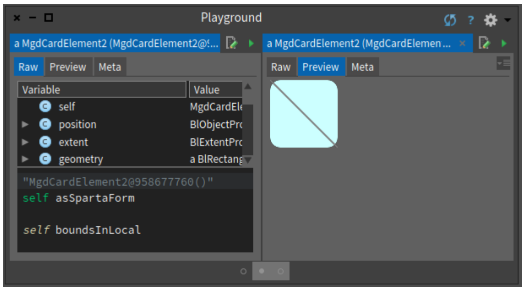
```
MgdCardElement >> drawPathOnSpartaCanvas: aCanvas
  super drawPathOnSpartaCanvas: aCanvas.
  self drawCommonOnCanvas: aCanvas.
  self card flipped
    ifTrue: [ self drawFlippedOnCanvas: aCanvas ]
    ifFalse: [ self drawBacksideOn: aCanvas ]
```

Now we are ready to implement the backside and flipped side

3.5 Adding a cross

Now we are ready to define the backside of our card. We will start by drawing a line. To draw a line we should provide a like as a path. In Bloc this can be done either that passing a Path object or asking the canvas for its shape factory. The shape factory encapsulate the logic of shapes. This is what we do below with the expression `path: (aCanvas shape line: 0 @ 0 to: self extent)`. The message `shape` returns a ShapeFactory and that ask this factory to produce a path to produce a line.

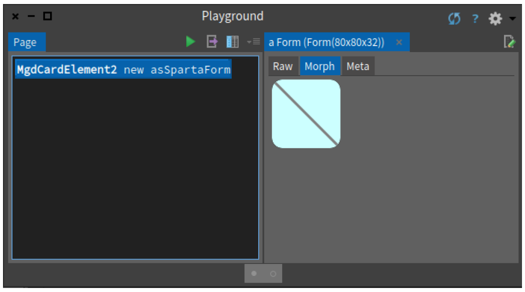
```
MgdCardElement >> drawBacksideOn: aCanvas
  aCanvas stroke
    paint: Color paleBlue;
    path: (aCanvas shape line: 0 @ 0 to: self extent);
    draw
```


Playground
a MgdCardElement2 (MgdCardElement2@9...
a MgdCardElement2 (MgdCardElemen...
Raw
Preview
Meta

Variable	Value
self	MgdCardEle
position	BlObjectPro
extent	BlExtentPro
geometry	a BlRectang

```
"MgdCardElement2@958677760()"
self asSpartaForm

self boundsInLocal
```

Figure 3-3 A rounded card with half of the cross.



```
MgdCardElement2 new asSpartaForm
```

Figure 3-4 Clipping line.

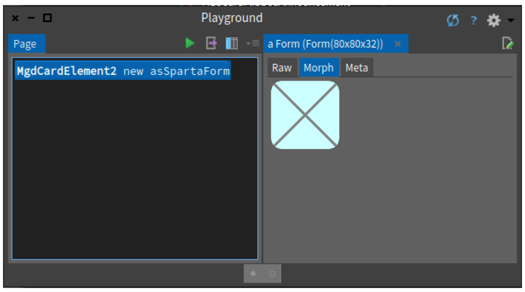
Once this method defined, refresh the inspector and you should get a card as in Figure 3-3.

3.6 Lines and corners reconciled

In Figure 3-3 we see that the line is not clipped to the corners. We should address this. This is the way we did it for now.

```
MgdCardElement >> drawBacksideOn: aCanvas
  | radiusOffset |
  radiusOffset := 12 / Float pi.
  aCanvas stroke
    paint: Color gray;
    width: 3;
    path: (aCanvas shape
        line: radiusOffset @ radiusOffset
        to: self extent - radiusOffset);
    draw
```

Once you change the method drawBacksideOn: and refresh you should get a card as displayed in Figure 3-4.



```
MgdCardElement2 new asSpartaForm
```

Figure 3-5 A card with a complete backside.

3.7 Full cross clipped

Now we can add the line to build a full cross. Our solution is defined as follows:

```
MgdCardElement >> drawBacksideOn: aCanvas
  | radiusOffset |
  radiusOffset := 12 / Float pi.
  aCanvas stroke
    paint: Color paleBlue;
    width: 3;
    path: (aCanvas shape
        line: radiusOffset @ radiusOffset
        to: self extent - radiusOffset);
    draw.
  aCanvas stroke
    paint: Color paleBlue;
    width: 3;
    path: (aCanvas shape
        line: (self width - radiusOffset) @ radiusOffset
        to: radiusOffset @ (self height - radiusOffset));
    draw
```

Now our backside is fully implemented and when you refresh your view, you should get the card as shown in Figure 3-5.

3.8 Flipped side

Now we are ready to develop the flipped side of the card. To see if we should change the card model. You can use the inspector to get the cardElement and send it the message `card flip` or directly recreate a new card as follows:

```
| cardEl |
cardEl := MgdCardElement2 new.
```

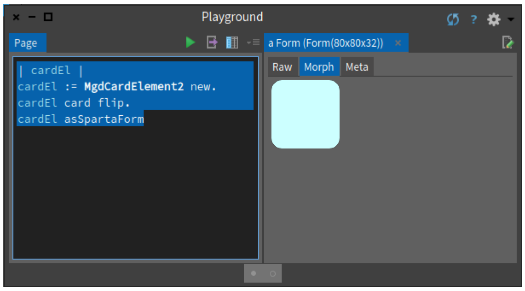

```
| cardEl |
cardEl := MgdCardElement2 new.
cardEl card flip.
cardEl asSpartaForm
```

Figure 3-6 A card with a complete backside.

```
cardEl card flip.
cardEl asSpartaForm
```

You should get an inspector in the situation shown in Figure 3-6. Now we are ready to implement the flipped side.

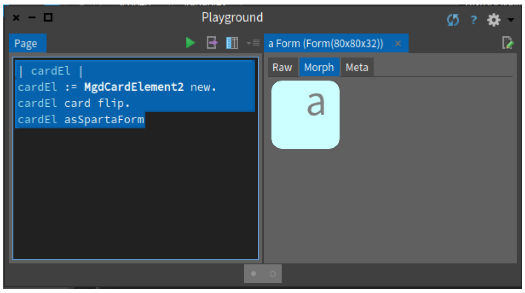
Let us redefine `drawFlippedOnCanvas:` as follows:

- First we ask the canvas to build font of 50. Note that for the font we specify a FreeType font (pay attention that strike font do not work and will never work in Bloc - in fact they will be removed once Pharo will be based on Bloc).
- Then we ask the canvas to draw a text using the font with the color we want.

We should not forget to send the message `draw` to the canvas.

```
MgdCardElement >> drawFlippedOnCanvas: aCanvas
  | font |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.
  aCanvas text
    font: font;
    paint: Color gray;
    string: self card symbol asString;
    draw
```

When we refresh the display we do not see the symbol and this is a problem. If you pay attention you will see that there is just one line that is drawn on the top left of the card. You can change the color to red to see it on the card. We are drawing the string in the corner and outside the rounded rectangle. Let us fix that issue by defining the baseline from which the text should be displayed.



```
| cardEl |
cardEl := MgdCardElement2 new.
cardEl card flip.
cardEl asSpartaForm
```

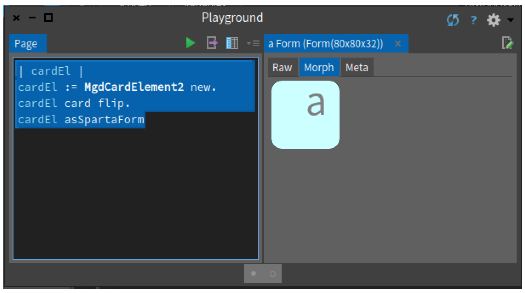
Figure 3-7 Not centered letter.

```
MgdCardElement >> drawFlippedOnCanvas: aCanvas
  | font origin |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.
  origin := self extent / 2.
  aCanvas text
    baseline: origin;
    font: font;
    paint: Color paleBlue;
    string: self card symbol asString;
    draw
```

When you refresh the inspector you should see the card symbol but not centered as shown in Figure 3-8.

To center well the text, we have to use exact font metrics. Bloc can support multiple graphical back-end such as Cairo, Moz2D and in the future plain openGL. There is one important constraints is that font metrics should be measured and manipulated via the same back-end abstraction. For this purpose, the expression `aCanvas text` returns a text painter and such a text painter provide access to the font measures. Using such measure we can then get access to the text metrics and compute a better center.

```
MgdCardElement >> drawFlippedOnCanvas: aCanvas
  | font origin textPainter metrics |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.

  textPainter := aCanvas text
    font: font;
    paint: Color paleBlue;
    string: self card symbol asString.
```



```
| cardEl |
cardEl := MgdCardElement2 new.
cardEl card flip.
cardEl asSpartaForm
```

Figure 3-8 Not centered letter.

```
  metrics := textPainter measure.

  origin := (self extent - metrics textMetrics bounds extent) / 2.
  textPainter
    baseline: origin;
    draw
```

With this definition we get the letter centered vertically but not horizontally. This is because we have to tak into accoun the font size.

```
MgdCardElement >> drawFlippedOnCanvas: aCanvas
  | font origin textPainter metrics |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.

  textPainter := aCanvas text
    font: font;
    paint: Color paleBlue;
    string: self card symbol asString.

  metrics := textPainter measure.

  origin := (self extent - metrics textMetrics bounds extent) / 2.
  origin := origin - metrics textMetrics bounds origin.
  textPainter
    baseline: origin;
    draw
```

With this definition we get a centered letter as shown in Figure 3-8.

Now we are ready to work on the board game.

CHAPTER 4

Adding a board view

In the previous chapter, we defined all the card visual. We are now ready to define the game board visual. Basically we will define a new element subclass and set its layout

Here is a typical scenario to create the game: we create a model, and its view and we assign the model as the view's model.

```
game := MgdGameModel numbers.
grid := MgdGameElement2 new.
grid memoryGame: game.
```

4.1 The GameElement class

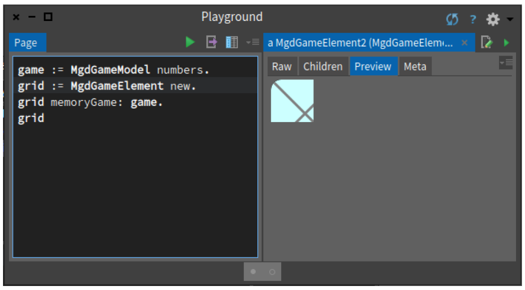
Let us define the class `MgdGameElement` that will represent the game board. As for the `CardElement`, it inherits from the `BlElement` class. This view object holds a reference to the game model.

```
BlElement subclass: #MgdGameElement
  instanceVariableNames: 'memoryGame'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Tutorial'
```

We define the `memoryGame:` setter method. We will extend it just after to create all the cards element.

```
MgdGameElement >> memoryGame: aMgdGameModel
  memoryGame := aMgdGameModel
```

```
MgdGameElement >> memoryGame
  ^ memoryGame
```



```
game := MgdGameModel numbers.
grid := MgdGameElement new.
grid memoryGame: game.
grid
```

Figure 4-1 A first board - not really working.

During the object initialization we set the layout (i.e., how sub elements are placed inside their container). Here we define the layout to be a grid layout and we set it as horizontal.

```
MgGameElement >> initialize
  super initialize.
  self layout: BlGridLayout horizontal.
```

4.2 Creating cards

When a model is set for a board game, we use the model information to perform the following actions:

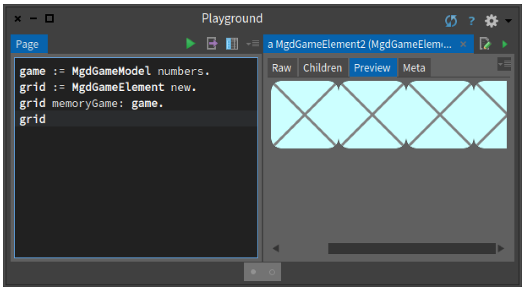
- we set the number of column of the layout
- we create all the card elements paying attention to set their respective model.

Note in particular that we add all the card graphical element as children of the board game using the message `addChild:`.

```
MgdGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel.

  memoryGame availableCards
    do: [ :aCard | self addChild: (self newCardElement card: aCard) ]
```

```
MgdGameElement >> newCardElement
  ^ MgdCardElement new
```

When we refresh the inspector we obtain a situation similar to the one of Figure 4-1. It shows that only a small part of the game is displayed. This is due to the fact that the game element did not adapt to its children.



```
game := MgdGameModel numbers.
grid := MgdGameElement new.
grid memoryGame: game.
grid
```

Figure 4-2 Displaying a row.

4.3 Updating the container to its children

A layout is responsible for the layout of the children of a container but not of the container itself. For this, we should use constraints.

```
MgdGameElement >> initialize
  super initialize.
  self layout: BlGridLayout horizontal.
  self
    constraintsDo: [ :layoutCons |
      layoutCons horizontal fitContent.
      layoutCons vertical fitContent ]
```

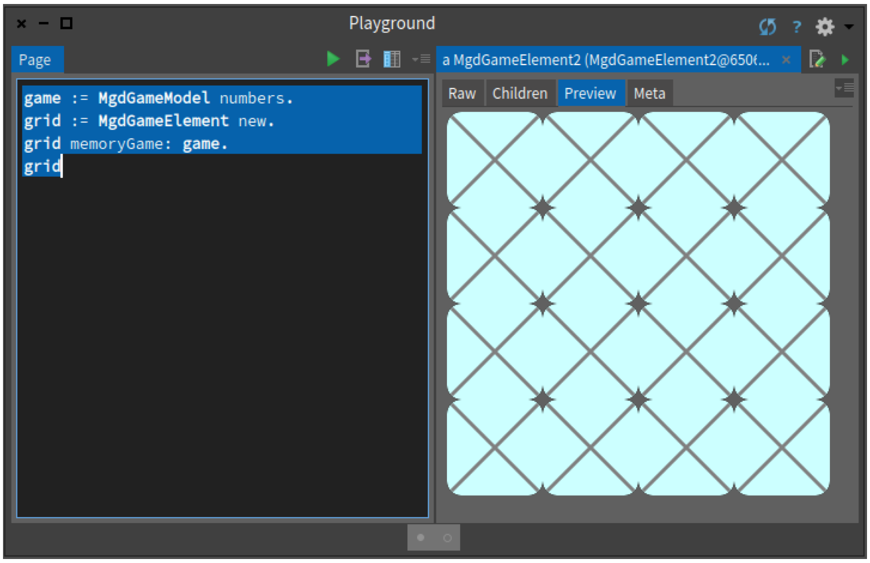
Now when we refresh our view we should get a situation close to the one presented in Figure4-2, i.e., having just one row. Indeed we did never mentioned to the layout that it should layout its children according to a grid wrapping after four.

4.4 Getting all the children displayed

We modify the `memoryGame:` method to set the number of columns that the layout should handle.

```
MgdGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel.
  self layout columnCount: memoryGame gridSize.
  memoryGame availableCards
    do: [ :aCard | self addChild: (self newCardElement card: aCard) ]
```

Once the layout is set with the correct information we obtain a full board as shown in Figure 5-1.



```
game := MgdGameModel numbers.
grid := MgdGameElement new.
grid memoryGame: game.
grid
```

Figure 4-3 Displaying a full board.

4.5 Separating cards

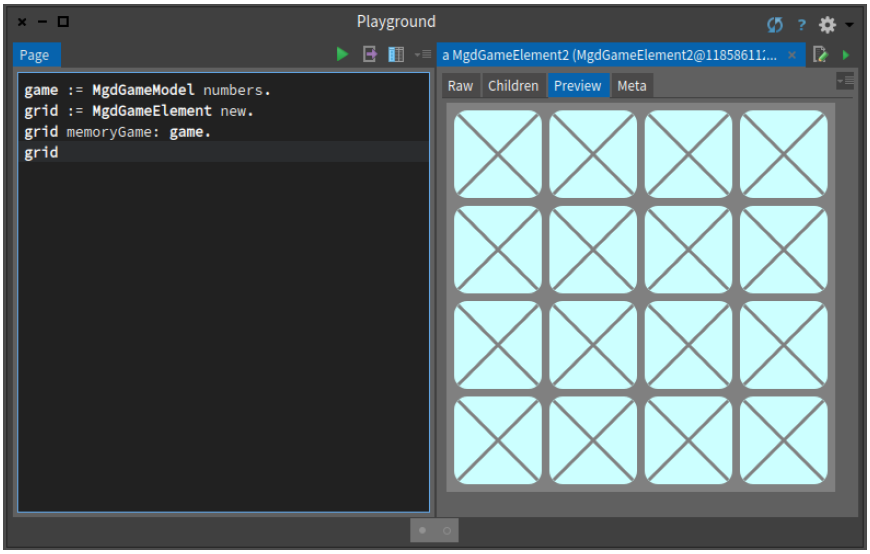
To offer a better identification of the cards, we should add some space between each of them. We achieve this by using the message `cellSpacing:` as shown below.

We take the opportunitu to change the background color using the message `background:`. Note that a background is not necessarily a color but that color is polymorphic to a color therefore the expression `background: Color gray` is equivalent to `background: (BlBackground fill: Color gray)`.

```
MgdGameElement >> initialize
  super initialize.
  self layout: BlGridLayout horizontal.
  self layout cellSpacing: 7.
  self background: (BlBackground fill: Color gray).
  self
    constraintsDo: [ :layoutCons |
      layoutCons horizontal fitContent.
      layoutCons vertical fitContent ]
```

Once this method changed, you should get a situation similar to the one described by Figure 4-4.

We are now eady for adding interation to the game.



```
game := MgdGameModel numbers.
grid := MgdGameElement new.
grid memoryGame: game.
grid
```

Figure 4-4 Displaying a full board with space.

CHAPTER 5

Adding Interaction

Now we will add interation to the game. We want to flip the card by clicking on them. Bloc supports such situation using two mechanisms: on one hand, event listeners handle events and on the other hand, the communication between the model and view is managed via the registration to announcements raised by the model.

5.1 An event listener

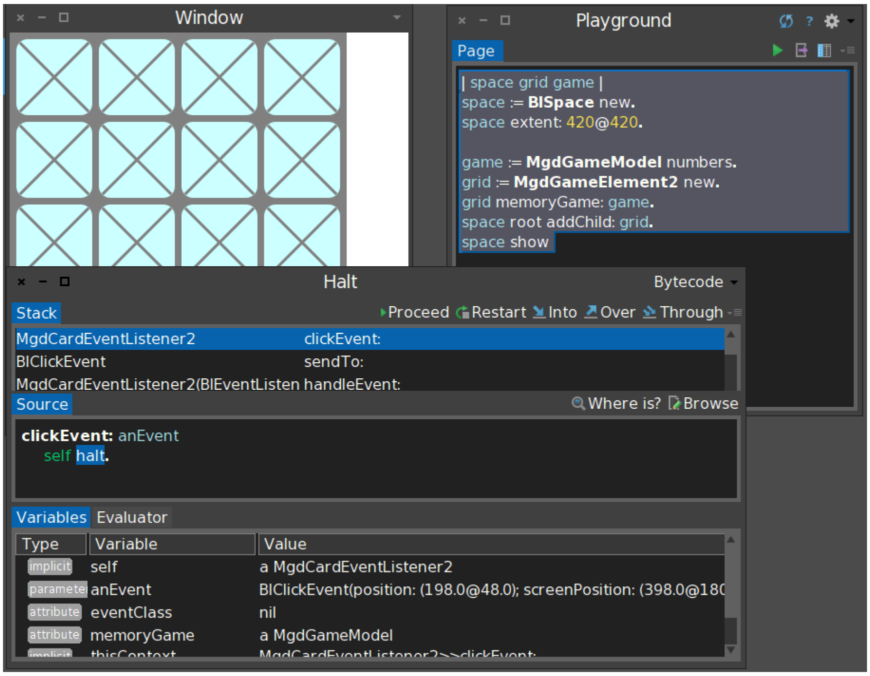
```
BlElementEventListener subclass: #MgdCardEventListener
  instanceVariableNames: 'memoryGame'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Tutorial'
```

We add an instance variable `memoryGame` holding a game model to the listener because we will need to access the model to react to event for example to update the game situation.

```
MgdCardEventListener >> memoryGame: aGameModel
  memoryGame := aGameModel
```

Let us redefine the `click:` method to raise a debugger. It will give us the occasion to introspect the system.

```
MgdCardEventListener >> clickEvent: anEvent
  self halt
```



```
| space grid game |
space := BlSpace new.
space extent: 420@420.

game := MgdGameModel numbers.
grid := MgdGameElement2 new.
grid memoryGame: game.
space root addChild: grid.
space show
```

```
MgdCardEventListener2                    clickEvent:
BlClickEvent                             sendTo:
MgdCardEventListener2(BlEventListen      handleEvent:
```

```
clickEvent: anEvent
    self halt.
```

Type	Variable	Value
implicit	self	a MgdCardEventListener2
paramete	anEvent	BlClickEvent(position: (198.0@48.0); screenPosition: (398.0@18(
attribute	eventClass	nil
attribute	memoryGame	a MgdGameModel
implicit	thisContext	MgdCardEventListener2>>clickEvent:

Figure 5-1 Debugging the clickEvent: anEvent method.

5.2 Adding event listeners

Now we should add the game event listener to each card because we want to know which card will be clicked and pass this information to the game model.

```
MgdGameElement >> newCardEventListener
  ^ MgdCardEventListener new
```

```
MgdGameElement >> memoryGame: aGameModel
  memoryGame := aGameModel.
  self layout columnCount: memoryGame gridSize.
  memoryGame availableCards
    do: [ :aCard |
      | cardElement |
      cardElement := self newCardElement card: aCard.
      cardElement addEventHandler: (self newCardEventListener
    memoryGame: aGameModel).
      self addChild: cardElement ]
```

Now the preview is not enough and we should create a window and embedded the game element. Then when you click on an card you should get a debugger as shown in Figure 5-1.

```
| space grid game |
space := BlSpace new.
space extent: 420@420.
game := MgdGameModel numbers.
grid := MgdGameElement2 new.
grid memoryGame: game.
space root addChild: grid.
space show
```

5.3 Specialize clickEvent:

Now we can specialise the `clickEvent:` method as follows:

- we get the graphical element that receives the mouse click using the message `currentTarget`. The message `currentTarget` returns the element that receives an event.
- From this graphical card we access the card model and we pass this card model to the game model.

```
MgdCardEventListener >> clickEvent: anEvent
  memoryGame chooseCard: anEvent currentTarget card
```

It means that the memory game model is changed but we do not see the visual effect of our actions. Indeed this is normal. We never made sure that visual elements are listening to model changes. This is what we will do in the following chapter.

5.4 Connecting the model to the UI

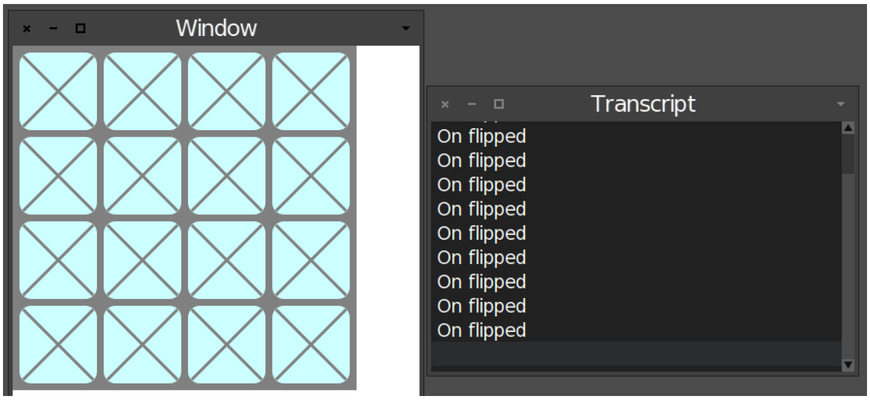
Now we show how the domain communicates with the user interface: the domain emit notifications using announcements but its does not refer to the UI elements. It is the visual elements that should register to the notifications and react accordingly.

Let us first define two simple methods in the class `CardElement` just producing a trace.

```
MgdCardElement >> onDisappear
  Transcript show: 'On disappear'; cr
```

```
MgdCardElement >> onFlipped
  Transcript show: 'On flipped'; cr
```

Now we can modify the setter so that when a card model is set to a card graphical element, we register to the notifications emitted by the model. In the following method, we make sure that on notifications we invoke the trace methods just defined.



```
On flipped
On flipped
On flipped
On flipped
On flipped
On flipped
On flipped
On flipped
On flipped
```

Figure 5-2 Tracing registration to the domain notifications.

```
MgdCardElement >> card: aMgCard
  card := aMgCard.
  card announcer when: MgdCardFlippedAnnouncement send: #onFlipped
    to: self.
  card announcer when: MgdCardDisappearAnnouncement send:
    #onDisappear to: self
```

Now when you click on a card, you can see the trace in the Transcript but you do not see the changes. This is because we should notify the graphics engine that one element should be redrawn.

```
MgdCardElement >> onFlipped
  Transcript show: 'On flipped'; cr.
  self invalidate
```

5.5 Handling disappear

There are two ways to implement the disappear of a card either setting the opacity of the element to 0.

```
MgdCardElement >> onDisappear
  Transcript show: 'On disappear'; cr.
  self opacity: 0.
  self invalidate
```

Note that the element is still present and receive events.

Or changing the visibility as follows:

```
MgdCardElement >> onDisappear
  Transcript show: 'On disappear'; cr.
  self visibility: BlVisibility hidden.
  self invalidate. "not needed in the latest Bloc"
```

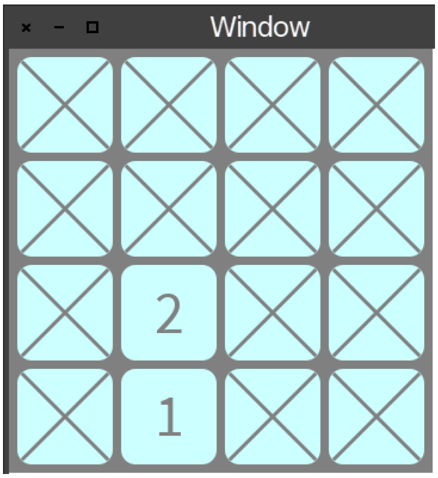


Figure 5-3 Selecting two cards that are not in pair.

Note that in such case, the element does not get events. It is used for layout.

5.6 Refreshing on missed pair

When the player selects two cards that are not in pair, we present the two cards as shown in Figure 5-3. Now the clicking on another card will flip back the previous cards.

Remember a card when flipped in either sense will raise a notification.

```
MgdCardElement >> flipped: aBoolean
  flipped := aBoolean.
  self notifyFlipped
```

In the method `choseCard:` we see that all hte previous cards are flip (toggled).

```
...
(self chosenCards size > self matchesCount)
  ifTrue: [
    self chosenCards allButLastDo: [ :each | each flip ].
    self chosenCards removeAll.
    self chosenCards add: aMgCard ]
...
```

5.7 Main interaction done

At the stage you are done for the simple interaction. The following chapter will explain how we can add animation.

CHAPTER 6

Adding animation