Building a memory game with Bloc

Andrei Chiș, Stéphane Ducasse and Aliaksei Syrel

November 9, 2017
master @ 5249447

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 shows 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.

Our goal is to have a functional game with the model and simple graphical user interface. In the end, the following code should be able to build, initialise and launch the game:

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

space := BlSpace new.
space extent: 420@420.
```

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.

```
space root addChild: grid.
space show
```

- first, we create a game 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 6.1 (`https://pharo.org/download`) running on the latest Pharo6.1 Virtual machine. You can get them at the following address

```
http://get.pharo.org/61+vm
```

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

```
wget -O- get.pharo.org/61+vm | bash
```

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

To load Bloc execute the following snippet in Pharo Playground:

```
Metacello new
   baseline: 'Bloc';
   repository: 'github://pharo-graphics/Bloc:pharo6.1/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. On 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 Bloc, 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 ]
```

2.2 Card simple operations

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

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

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

2.3 Adding notification

The notification is implemented 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 to these announcements as we will see later.

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

```
MgdCardModel >> notifyDisappear
  self announcer announce: MgdCardDisappearAnnouncement new
```

Here, `MgdCardFlippedAnnouncement` and `MgdCardDisappearAnnouncement` 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: $)
```

2.4 Reviewing the game model

The game model is simple: it keeps the tracks of 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'
```

The `initialize` method sets two collections for the different cards.

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

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

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

2.5 Grid size and card number

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
```

2.6 Initialization

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 shuffles it. We also add an assertion in this method to verify that the caller 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 |
       MgdCardModel new symbol: aSymbol ] ])
          flattened shuffled asOrderedCollection
```

2.7 Game logic

Next we need `chooseCard:`, a method that will be called when a user selects 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 view uses animations that give players the chance 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 amount of cards and they all show the same symbol. In this case, all selected cards receive the message `disappear` 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 situation the two initial cards will be flipped back. The list of selected 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
```

2.8 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 a subclass of `BlElement`, the inheritance tree root. In subsequent chapters we will do the same for the game and add interaction using 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: #MgdRawCardElement
  instanceVariableNames: 'card'
  classVariableNames: ''
  package: 'Bloc-MemoryGame-Demo-Elements'
```

The message `backgroundPaint` will be used later to customise the background of our card element. Let us define a nice color.

```
MgdRawCardElement >> backgroundPaint
  ^ Color lightGray
```

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

```
MgdRawCardElement >> card
  ^ card
```

```
MgdRawCardElement >> card: aMgCard
  card := aMgCard
```

We initialize it to get a

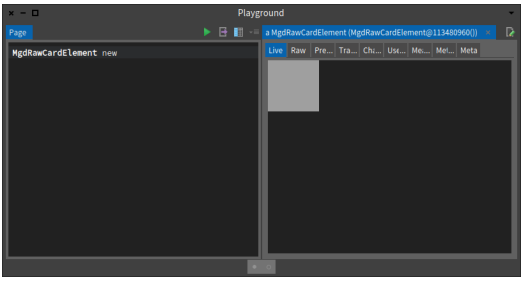

Figure 3-1 A first extremely basic representation of face down card.

```
MgdRawCardElement >> initialize
  super initialize.
  self size: 80 @ 80.
  self card: (MgdCardModel new symbol: $a)
```

3.2 Starting to draw a card

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

```
MgdRawCardElement >> drawOnSpartaCanvas: aCanvas
  aCanvas fill
    paint: self backgroundPaint;
    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 can inspect our card element in Playground (`CMD+g`) and switch to `Live` presentation as shown in Figure 3-1:

```
MgdRawCardElement new
```

3.3 Improving the card visual

Instead of displaying a full rectangle, we want a better visual. `Sparta` canvas 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. Result is shown in Figure 3-2:

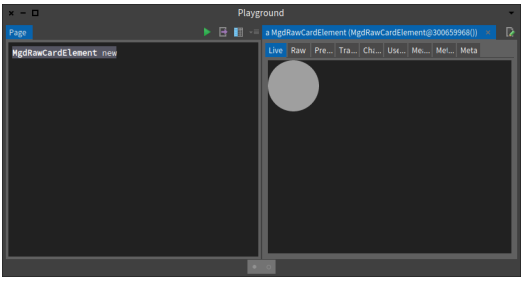

Figure 3-2 A card with circular background.

```
MgdRawCardElement >> drawOnSpartaCanvas: aCanvas
  aCanvas fill
    paint: self backgroundPaint;
    path: (aCanvas shape ellipse: self boundsInLocal);
    draw
```

However, we don't want the card to be a circle either. Ideally it should be a rounded rectangle. That is why let's first add a helper method that would provide us with a corner radius:

```
MgdRawCardElement >> cornerRadius
  ^ 12
```

Since for our card we would like to have a rounded rectangle so we use the `roundedRectangle:radii:` factory message. However, this time, instead of just directly drawing a rounded rectangle we will fill the whole card as we did on the first step with `background paint` and then simply `clip` everything by rounded rectangle:

```
MgdRawCardElement >> drawOnSpartaCanvas: aCanvas
  | roundedRectangle |

  roundedRectangle := aCanvas shape
    roundedRectangle: self boundsInLocal
    radii: (BlCornerRadii radius: self cornerRadius).

  aCanvas clip
    by: roundedRectangle
    during: [
      aCanvas fill
        paint: self backgroundPaint;
        path: self boundsInLocal;
        draw. ]
```

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

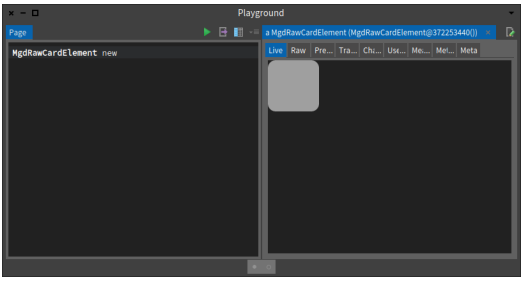

Figure 3-3 A rounded card.

3.4 Preparing flipping

We define now two methods

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

```
MgdRawCardElement >> drawFlippedSideOn: aCanvas
  "nothing for now"
```

And we refactor `drawOnSpartaCanvas:` as follows:

```
MgdRawCardElement >> drawOnSpartaCanvas: aCanvas
  | roundedRectangle |

  roundedRectangle := aCanvas shape
    roundedRectangle: self boundsInLocal
    radii: (BlCornerRadii radius: self cornerRadius).

  aCanvas clip
    by: roundedRectangle
    during: [
      aCanvas fill
        paint: self backgroundPaint;
        path: self boundsInLocal;
        draw.

      self card isFlipped
        ifTrue: [ self drawFlippedSideOn: aCanvas ]
        ifFalse: [ self drawBacksideOn: aCanvas ] ]
```

we extract the common part into a separate method.

```
MgdRawCardElement >> drawCommonOn: aCanvas
  aCanvas fill
    paint: self backgroundPaint;
    path: self boundsInLocal;
    draw
```

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

```
MgdRawCardElement >> drawOnSpartaCanvas: aCanvas
  | roundedRectangle |

  roundedRectangle := aCanvas shape
    roundedRectangle: self boundsInLocal
    radii: (BlCornerRadii radius: self cornerRadius).

  aCanvas clip
    by: roundedRectangle
    during: [
      self drawCommonOn: aCanvas.
      self card isFlipped
        ifTrue: [ self drawFlippedSideOn: 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 it as a path. In Bloc this can be done by either passing a Path object or by asking the canvas for its shape factory. The shape factory encapsulates 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 we ask this factory to produce a line path.

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

Once this method is defined, refresh the inspector and you should get a card as in Figure 3-4.

3.6 Full cross

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

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

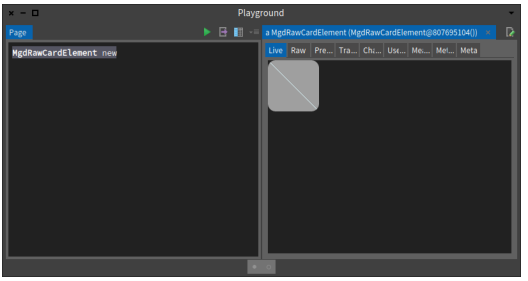

Figure 3-4 A rounded card with half of the cross.

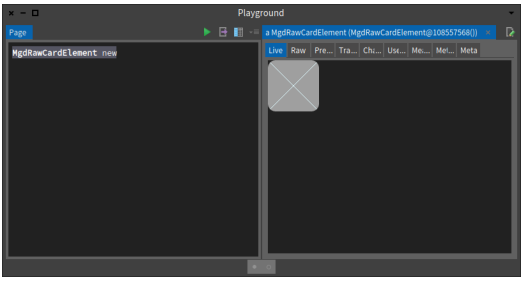

Figure 3-5 A card with a complete backside.

```
  aCanvas stroke
    paint: Color paleBlue;
    path: (aCanvas shape line: self width @ 0 to: 0@self height);
    draw
```

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

3.7 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:

```
| cardElement |
cardElement := MgdRawCardElement new.
cardElement card flip.
cardElement
```

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

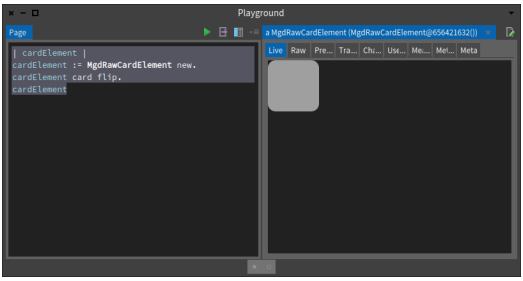

Figure 3-6 A flipped card without any visuals.

Let us redefine `drawFlippedSideOn:` as follows:

- First we ask the canvas to build a font of size 50. Note that for the font we specify a FreeType font (pay attention that strike fonts do not work and will never work in Bloc - in fact they will be removed once Pharo is 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.

```
MgdRawCardElement >> drawFlippedSideOn: aCanvas
  | font |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.
  aCanvas text
    font: font;
    paint: Color white;
    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.

```
MgdRawCardElement >> drawFlippedSideOn: aCanvas
  | font origin |
  font := aCanvas font
    named: 'Source Sans Pro';
    size: 50;
    build.
```

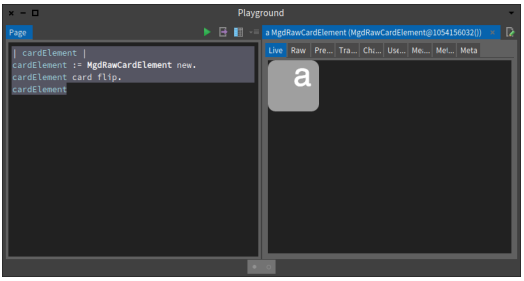

Figure 3-7 Not centered letter.

```
  origin := self extent / 2.0.
  aCanvas text
    baseline: origin;
    font: font;
    paint: Color white;
    string: self card symbol asString;
    draw
```

When you refresh the inspector you should see the card symbol but not centered as shown in Figure 3-7.

To center the text well, 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 constraint, that 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 provides access to the font measurements. Using such measurements we can then get access to the text metrics and compute a better center.

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

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

  metrics := textPainter measure.

  origin := (self extent - metrics textMetrics bounds extent) / 2.0.
  textPainter
    baseline: origin;
```

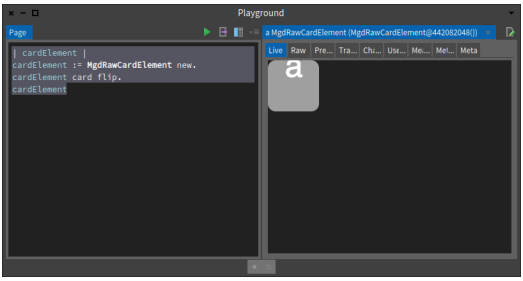

Figure 3-8 Horizontally centered letter.

```
    draw
```

With this definition we get the letter centered horizontally but not vertically as shown in Figure 3-8. This is because we have to take into account the font size.

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

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

  metrics := textPainter measure.

  origin := (self extent - metrics textMetrics bounds extent) / 2.0.
  origin := origin - metrics textMetrics bounds origin.
  textPainter
    baseline: origin;
    draw
```

With this definition we get a centered letter as shown in Figure 3-9.

Now we are ready to work on the board game.

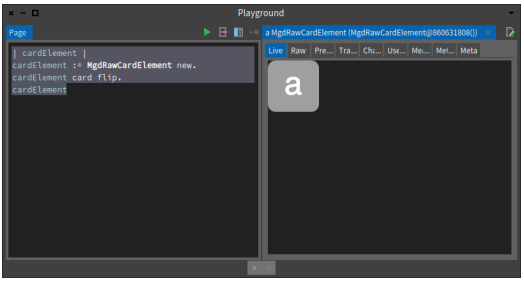

Figure 3-9 Not centered letter.

CHAPTER 4

Adding a board view

In the previous chapter, we defined all the card visualization. We are now ready to define the game board visualization. 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 := MgdGameElement new.
grid memoryGame: game.
```

4.1 The GameElement class

Let us define the class `MgdGameElement` that will represent the game board. As for the `MgdRawCardElement`, 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-Demo-Elements'
```

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
```

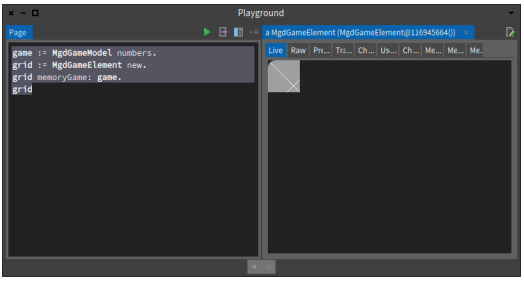

Figure 4-1 A first board - not really working.

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.

```
MgdGameElement >> 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 columns of the layout
- we create all the card elements paying attention to set their respective model.

Note in particular that we add all the cards graphical elements 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
  ^ MgdRawCardElement 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.

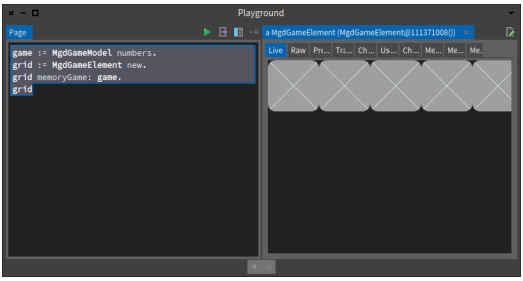

Figure 4-2 Displaying a row.

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: [ :aLayoutConstrants |
      aLayoutConstraints horizontal fitContent.
      aLayoutConstraints 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 never mentioned to the layout that it should layout its children into 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.

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

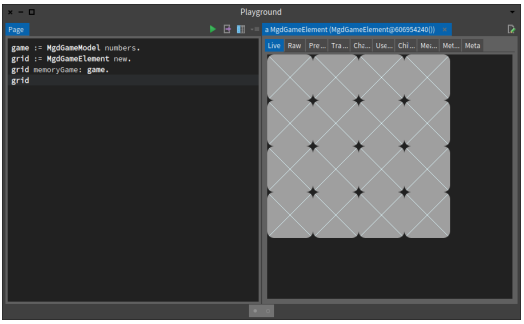

Figure 4-3 Displaying a full board.

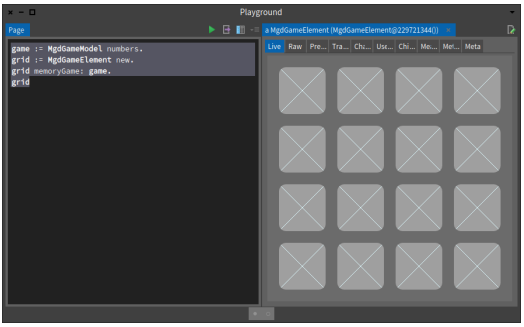

Figure 4-4 Displaying a full board with space.

shown below.

We take the opportunity 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 background therefore the expression `background: Color gray darker` is equivalent to `background: (BlBackground paint: Color gray darker)`.

```
MgdGameElement >> initialize
  super initialize.
  self background: (BlBackground paint: Color gray darker).
  self layout: (BlGridLayout horizontal cellSpacing: 20).
  self
    constraintsDo: [ :aLayoutConstraints |
      aLayoutConstraints horizontal fitContent.
      aLayoutConstraints vertical fitContent ]
```

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

We are now ready for adding interaction to the game.

CHAPTER 5

Adding Interaction

Now we will add interaction to the game. We want to flip the cards by clicking on them. Bloc supports such situations 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 sent by the model.

5.1 An event listener

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

We add an instance variable `memoryGame` holding a game model to the listener because we will need to access the model to react to events 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
```

5.2 Adding event listeners

Now we should add the card 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
```

For that we have to extend `#memoryGame:` model setter in `MgdGameElement` as follows by adding a card event listener to every card element using `#addEventHandler:`:

```
MgdGameElement >> memoryGame: aMgdGameModel
  | aCardEventListener |

  memoryGame := aMgdGameModel.
  aCardEventListener := self newCardEventListener memoryGame:
    aMgdGameModel.

  self layout columnCount: memoryGame gridSize.

  memoryGame availableCards
    do: [ :aCard |
      | cardElement |
      cardElement := self newCardElement card: aCard.
      cardElement addEventHandler: aCardEventListener.
      self addChild: cardElement ]
```

Please note, that in our case we can reuse the same event handler for all card elements. It allows us to reduce overall memory consumption and improve game initialisation time.

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

5.3 Specialize clickEvent:

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

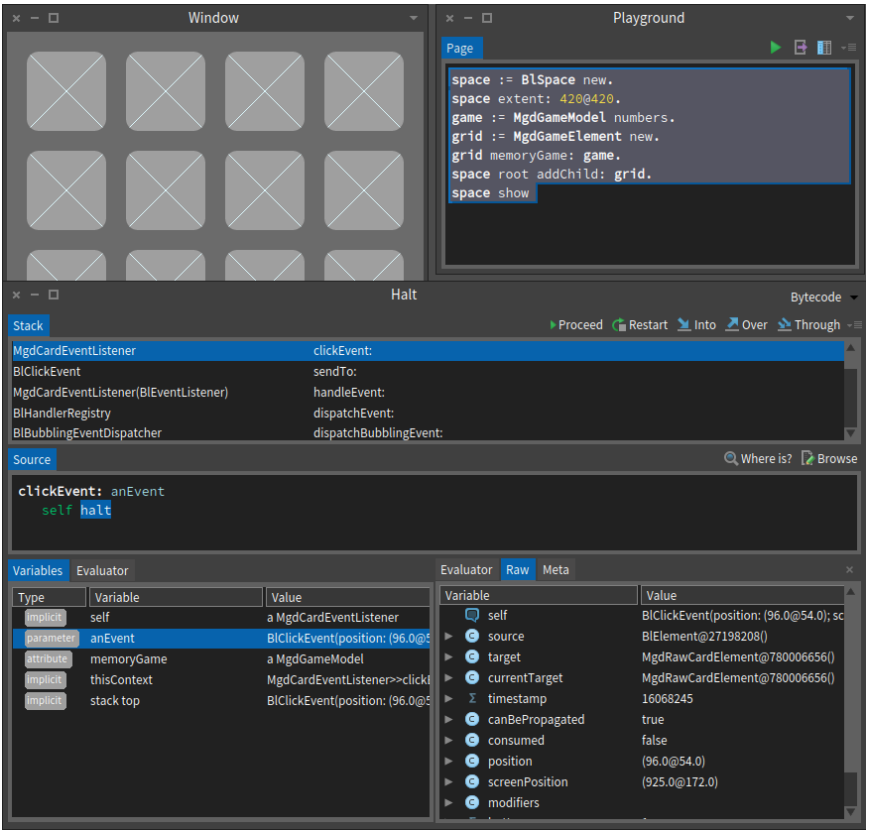

Figure 5-1 Debugging the clickEvent: anEvent method.

- 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.

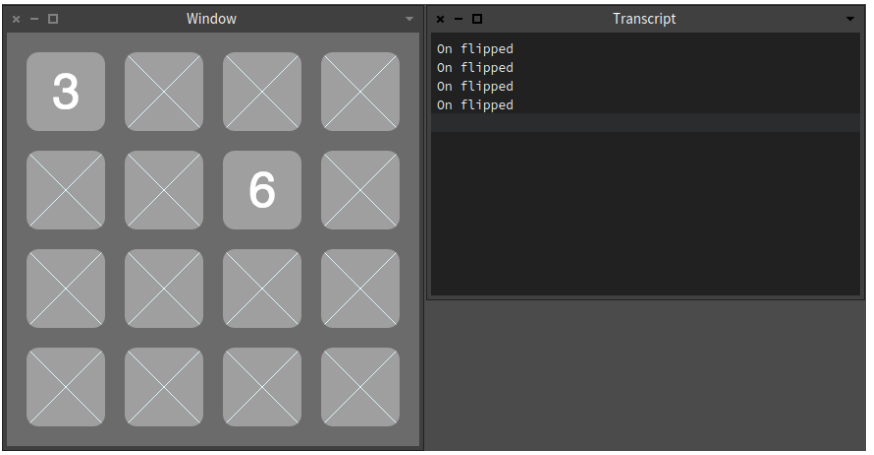

Figure 5-2 Tracing registration to the domain notifications.

5.4 Connecting the model to the UI

Now we show how the domain communicates with the user interface: the domain emits notifications using announcements but it 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 `MgdRawCardElement` just producing a trace.

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

```
MgdRawCardElement >> 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.

```
MgdRawCardElement >> 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.

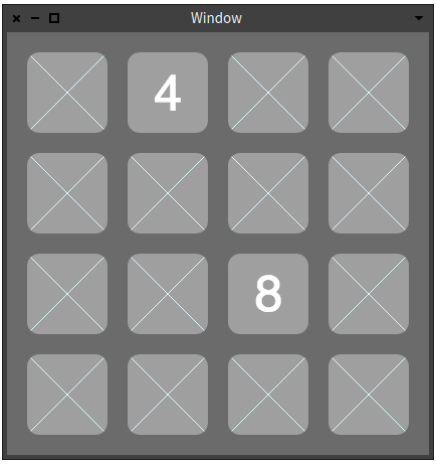

Figure 5-3 Selecting two cards that are not in pair.

```
MgdRawCardElement >> 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.

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

Note that the element is still present and receive events.

Or changing the visibility as follows:

```
MgdRawCardElement >> onDisappear
  Transcript show: 'On disappear'; cr.
  self visibility: BlVisibility hidden
```

Note that in the last 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 a pair, we present the two cards as shown in Figure 5-4. Now the clicking on another card will flip back the previous cards.

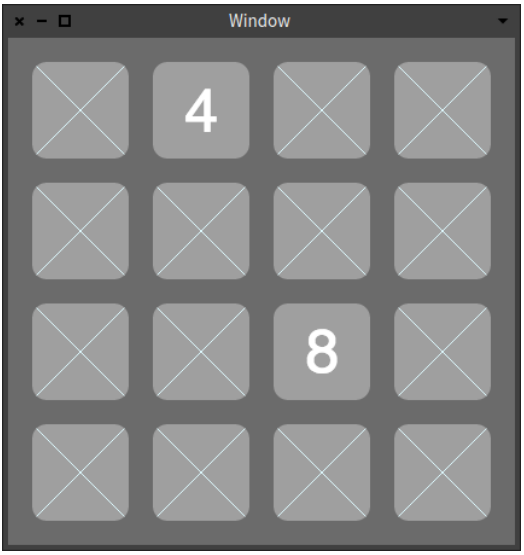

Figure 5-4 Selecting two cards that are not a pair.

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

```
MgdCardModel >> flip
  flipped := aBoolean.
  self notifyFlipped
```

In the method `#resetStep` we see that all the previous cards are flipped (toggled).

```
MgdGameModel >> resetStep
  | lastCard |

  lastCard := self chosenCards  last.

  self chosenCards
    allButLastDo: [ :aCard | aCard flip ];
    removeAll;
    add: lastCard
```

5.7 Conclusion

At this stage you are done for the simple interaction. Future versions of this document will explain how to add animations.