A “Fifteen Puzzle” in Fran

Conal Elliott

October, 1998

Technical Report

MSR-TR-98-54

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

A “Fifteen Puzzle” in Fran

Conal Elliott

`http://www.research.microsoft.com/~conal`

Microsoft Research

October, 1998

1. Introduction

This note describes a `Fran` implementation of the classic “Fifteen Puzzle”, and in doing so conveys by example the Fran style of programming interactive behavior [3][1]. See [2] for more discussion of this style and a comparison with more conventional programming styles, using the example of an interactive curve editor.

It is a good design practice is to decompose a problem into simpler pieces, solve the pieces independently, and compose the solutions. This practice, which is the essence of modularity, makes the tasks of programming and maintenance more manageable mentally, and sometimes gives rise to reusable components. As this note demonstrates, Fran promotes modularity by providing a high-level vocabulary in which to express the data exchanged by components of the overall solution. Fran’s host language, Haskell [5][7], provides complementary support for modularity, through polymorphism, higher-order functions, and laziness [6].

2. The Fifteen Puzzle

The Fifteen Puzzle, shown in Figure 1, was invented by Sam Loyd sometime around 1870. The object of the puzzle is to move the pieces around by sliding until they are in order, with the blank space being in the upper-right corner. If a piece is adjacent to the blank space, it may be moved there. For instance, in the configuration on the left half of the figure, the “1” piece may be moved east, the “3” piece south, the “8” west, or the “15” north. If one of these moves is made, then the blank spot and the moved piece trade locations.

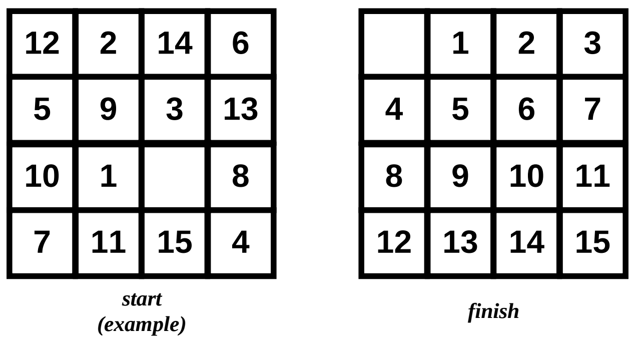

Figure 1. The Fifteen Puzzle

It turns out that exactly half of the possible initial configurations are solvable. Sam Loyd, being a practical joker and genius puzzler, offered $1000 to the first person to solve the puzzle from a configuration that was all ordered but with the “14” and “15” swapped. Loyd never had to pay, because this configuration is unsolvable [4].

3. Functional reactive animation

The Fifteen Puzzle, and indeed any interactive program, may be seen as having the overall structure shown in Figure 2. The details of the “input stream” and the “display stream” are specific to a given operating/windows system. The input stream contains event structures containing a integer to denote the type of event (button click, mouse motion, key press, window resize, etc.), together with a few fixed fields indicating mouse position, which key was pressed, etc. Under Windows®, the display stream consists of calls to GDI and DirectX (perhaps via a higher-level library).

Fran simplifies construction of interactive applications by providing higher-level abstractions to input and display. As illustrated in Figure 3, the box labeled "application" of Figure 2 is decomposed into three components, logically running in parallel.

"Input packaging" converts the raw input stream into a higher-level, operating-system-independent data type called "`User`", defined by Fran.[1] The `User` type separates the raw input stream into several independent, strongly typed event streams for more convenient use, and provides temporally continuous "behaviors" for quantities like mouse and stylus positions and window size.

"Display" converts from a high-level "`ImageB`" value into a stream of display commands that displays it. The `ImageB` type (for "image behavior") represents a time-varying image (or "animation"), as a self-contained first-class value.

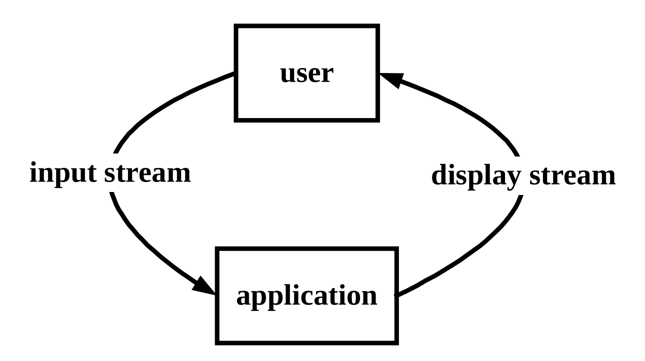

Figure 2. Interactive application

Between input packaging and display, lies the interactive animation itself, which maps from a `User` value to an `ImageB` value.

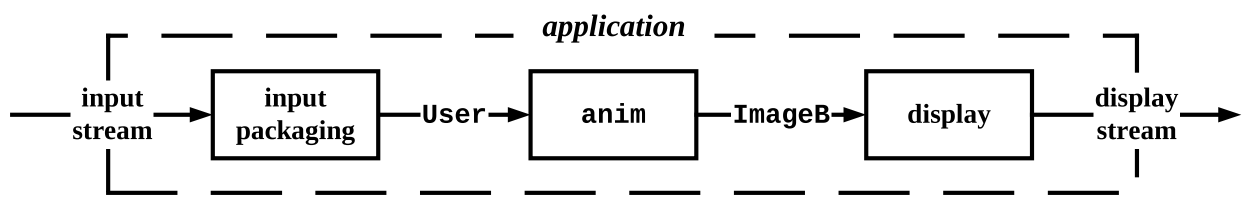

Figure 3 . Fran application

A benefit of decomposing the interactive application into these three components is that the first and third pieces are independent of a particular application, and so are provided by Fran. The application writer can then focus his or her effort on the content of the application, rather than details of input capture and display generation. Note also that the remaining component requires no operating-system-specific programming.

4. User interface vs. model

The decomposition in the previous section is provided by Fran. Another generally applicable decomposition may be applied to the interactive animation component:

- conversion of user input into "abstract input";
- an "interactive model" that maps abstract input to abstract output; and
- visualization of the abstract output over time, i.e., conversion into a 2D or 3D animation, possibly with sound

To apply this general decomposition, which may be termed "user interface vs. model", we first need to identify the types of the abstract input and output. In the case of the Fifteen Puzzle, we take as abstract input an event indicating a desired movement direction.[2] (Fran events are streams of "occurrences", each of which is a time-stamped value.) In Fran, this type is called "`Event Dir`", meaning an event each of whose occurrences contains a value of type `Dir`. The `Dir` type is defined for this particular application. It is a simple enumerated type containing the four values `East`, `West`, `North`, and `South`.

[1] The "`User`" type is somewhat misnamed, since it represents only the input coming from the user into an interactive animation.

[2] This definition of abstract input is not the only possibility. An alternative is a location-valued event telling what to move.

For the abstract output, we will use a list of time-varying puzzle piece "locations". This type is written "`[LocB]`", where `LocB` is a time-varying puzzle piece location, and the brackets indicate a list. The static (non-time-varying) location type is called `Loc`, and is simply a pair of integers, indicating column and row numbers respectively.

Given these choices of abstract input and output types, the UI-vs-model decomposition of the puzzle is shown in Figure 4. Note again that the three components logically work concurrently. The corresponding Haskell code for this visual definition, and all of those given below, are given in Appendix A.

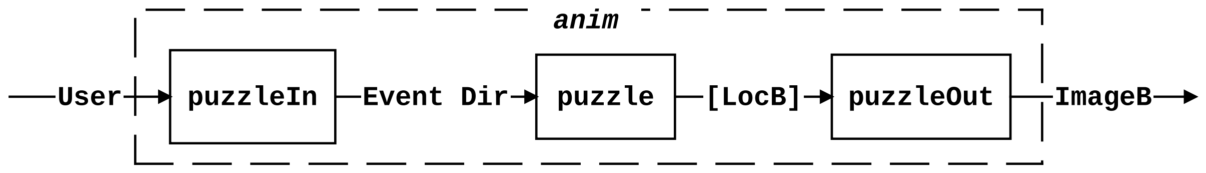

Figure 4. UI-vs-model decomposition

5. Model decomposition

In the previous section, we broke the interactive animation into three components, which can then be implemented in any order. We turn our attention first to the "model", i.e., `puzzle`.

Because the workings of the abstract puzzle are somewhat complicated, we would like to decompose it even further. This time we have no generally applicable guidelines to offer. Considering the nature of the Fifteen Puzzle, however, suggests one obvious breakdown:

- the movement of a single puzzle piece; and
- the combination of fifteen individual pieces into a whole puzzle

Unlike the UI-vs-model decomposition, this one involves replication of one component. Another important difference is that the two components feed information to each other, rather than being composed in a directed pipeline. The combination clearly depends on the individual pieces. The reason for the reverse dependency is that each piece needs to know the location of the blank spot, and that location is influenced by the actions of each piece.

Consider first the activity of a single puzzle piece, as computed by a component called `onePiece`, whose signature (inputs and outputs) are shown in Figure 5. A piece needs to know the blank's location at the time of a move request in order to decide whether to move. The piece's move request must then contain the blank location in addition to the desired direction.[3] It also needs to know where to start. In order for the puzzle to track the blank spot, each piece yields not only its time-varying location, but also an event indicating when and from where the piece moved.[4]

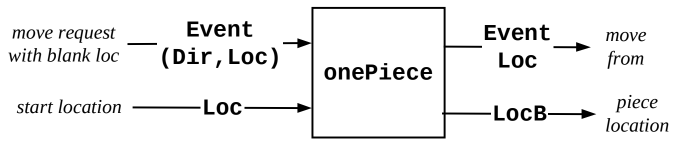

Figure 5. Single piece (signature)

We can construct the input event needed for `onePiece` by snapshotting the location of the blank whenever a directional move request is made. Doing so requires a means of tracking the blank, as supplied by the component `trackBlank,` whose signature is shown in Figure 6.

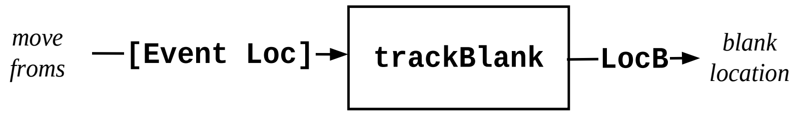

Figure 6. Tracking the blank (signature)

[3] The type `(Dir,Loc)` means the type of pairs `(d,l)`, where `d` is a direction and `l` is a (static) location.

[4] The "when" is implicitly part of all events, while the "from where" is explicit in the event type.

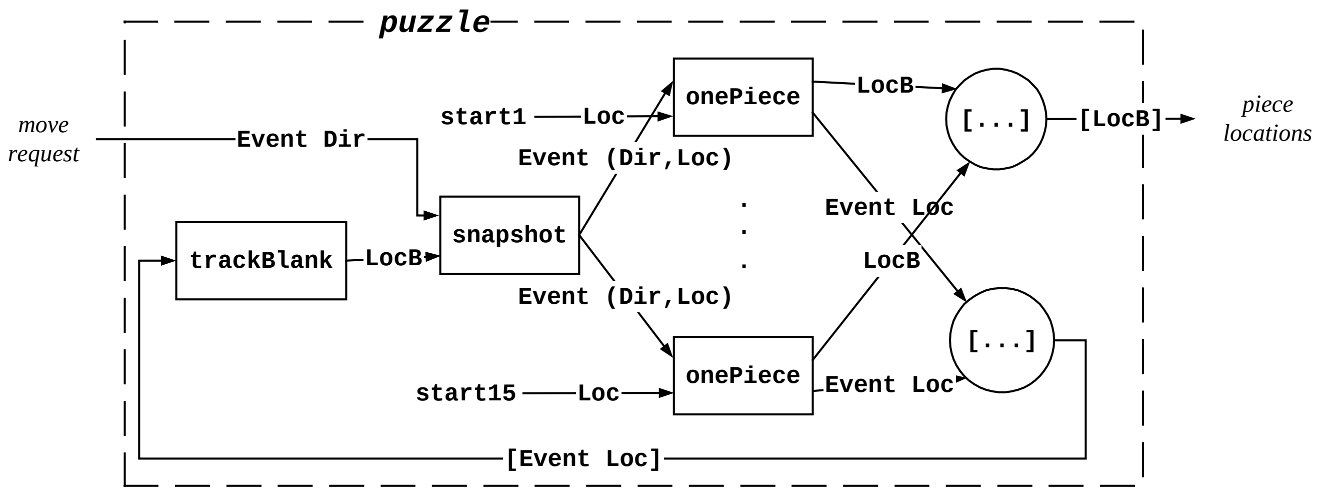

Figure 7. `puzzle` **decomposition**

To construct the puzzle, we place `trackBlank` into a feedback cycle with fifteen uses of `onePiece`, as illustrated in Figure 7. (The circular nodes containing "`[...]`" indicate list construction.)

6. One puzzle piece

The inner workings of `onePiece` are shown in Figure 8. In order for a single piece to move around, it compares its own location with the blank's when asked to move. The blank location is part of the move request, thanks to `trackBlank` and `snapshot` (see Figure 7) but the piece's own location must also be snapshotted. The resulting enriched event is "filtered" to accept only legal moves (as judged by `okMove`, defined in Appendix A), and to remove the direction, which is no longer useful. The result of filtering is an event containing the source and destination location, and indicating that the piece should move. This pair-valued move event is then "unzipped" into two separate events, containing just the source and just the destination. The former is used in `onePiece`'s result, and the latter is used to define the piece's location using `stepper`.

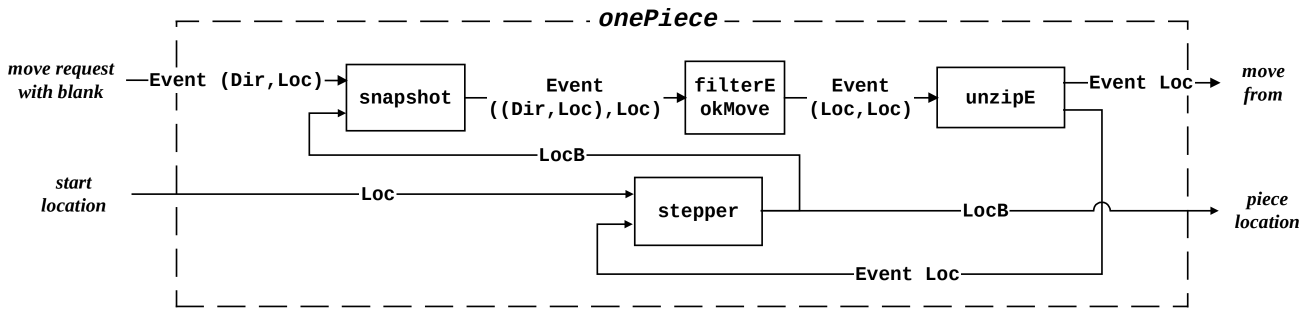

Figure 8. Activity of single puzzle piece

7. Tracking the blank

The blank-tracking component is shown in Figure 9. Whenever any piece moves *from* a location, the blank moves *to* that location. To track the location of the blank, therefore, we can use Fran's `anyE` event operator to merge all of the move-from events produced by `onePiece`, to form a single move-from event. From this combined event, which tells when and to where the blank should move, and the starting location of the blank space, we use Fran's `stepper` function to construct the blank's time-varying location. (In general, `anyE` combines a list of events $e_1,\ldots, e_n$ into a single event e that occurs exactly when any of the e_i occurs.)

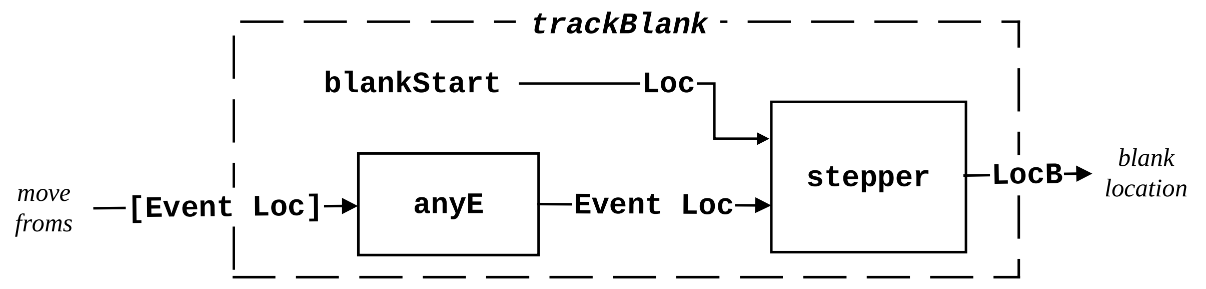

Figure 9. Tracking the blank piece

8. User Interface

Now we have finished with the abstract puzzle, but it still needs a user interface. Conversion of user input to abstract input (direction-valued events) is shown in Figure 10. Given a user value, `puzzleIn` first extracts the `keyPressAny` event, whose occurrence values contain a "virtual key code". This key-press event is filtered through a `keyDir` function, which maps each key code to the corresponding direction, if any. The up-arrow key maps to north, right-arrow to east, etc. Key presses other than the arrows yield no occurrence of the resulting direction-valued event.

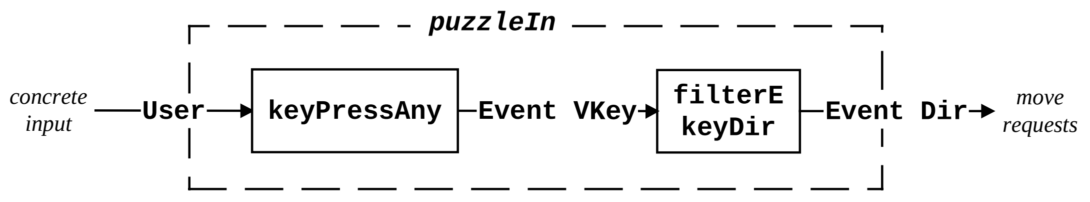

Figure 10. Conversion to abstract input

Finally, visualization of abstract puzzle output (lists of location behaviors) is given in Figure 11. Every piece is given a number image from the list `numIms`. That number image is moved to a position that corresponds to the piece's location, using the function `moveToLoc`. (The asterisk in the figure is intended to indicate that `moveToLoc` is applied to each corresponding location/image pair in the lists, and that the resulting images are collected into a result list.)

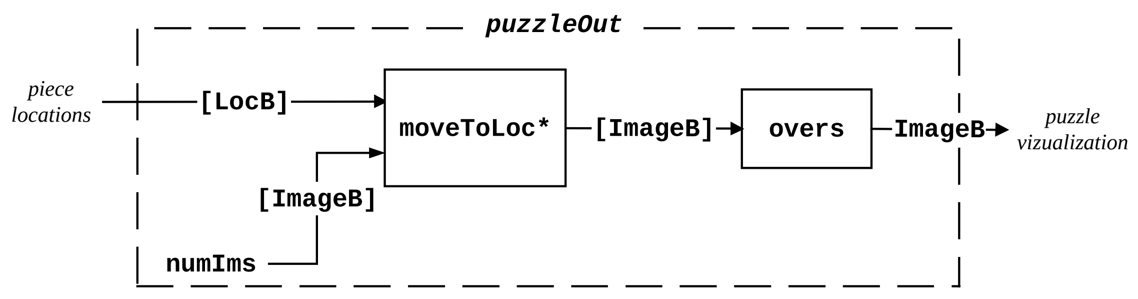

Figure 11. Visualization of abstract output

9. Input variations

The Fifteen Puzzle is not much of a puzzle until the pieces are mixed up. We could ask a user to do the mixing up, but why not *simulate* the user doing so? All we need is an another way to generate abstract input, i.e., an alternative to `puzzleIn`, having the same signature, but producing random moves. This new input component is shown in Figure 12. The resulting direction-valued event occurs at each "behavior update", a Fran event occurring roughly ten times per second. The (Haskell) function `random` takes an integer range, in this case zero to three, and a "seed" integer, and produces an infinite stream of pseudo-random integers. The Fran function `withElem_` takes an event and a list, and replaces the event's occurrence values with values from the list. Finally, the `toEnum` function is used to transform the integers to directions (because `Dir` is an enumeration type).

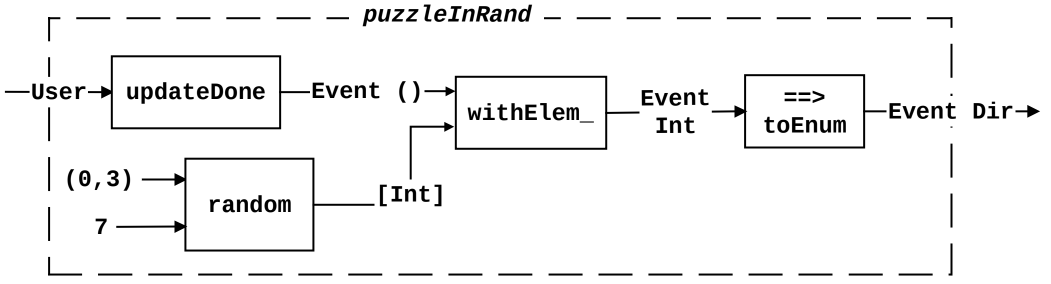

Figure 12. Random puzzle input

We could now replace `puzzleIn` with `puzzleInRand` in the definition `anim` (Figure 4), and the resulting puzzle would run entirely by itself, making random moves. Instead, we want the puzzle to switch to randomizing or normal mode whenever the user presses the '**r**' or space key, respectively. We achieve this behavior by yet another producer of abstract input, shown in Figure 13. The `changeMode` component, defined below, produces an event whose occurrence values are direction-valued events, indicating to change input modes. The `switcher` function takes an initial mode and the "mode-valued" event generated by `changeMode`, and produces a new direction event that starts out behaving like puzzleIn and changes whenever `changeMode` comes up with a new mode.

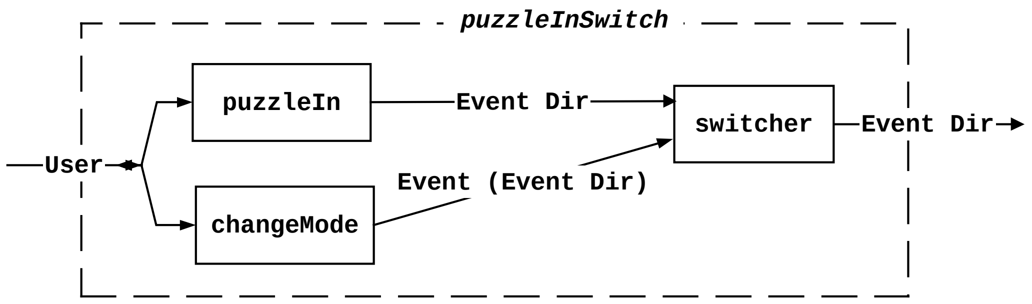

Figure 13. Switching between random and normal input modes

The mode-changing component is shown in Figure 14. It uses a function `charPressU` that makes an event indicating when a given character is pressed on the user's keyboard. The occurrence values contain a "residual user", which is passed through `puzzleInRand` if the '**r**' key is pressed, or through `puzzleIn` if the space key is pressed. The two resulting events, whose values are input modes (direction-valued events) are then merged into a single event of the same type, with Fran's "`.|.`" operator, which is the binary version of the function `anyE`, used in defining `puzzle` (Figure 7).

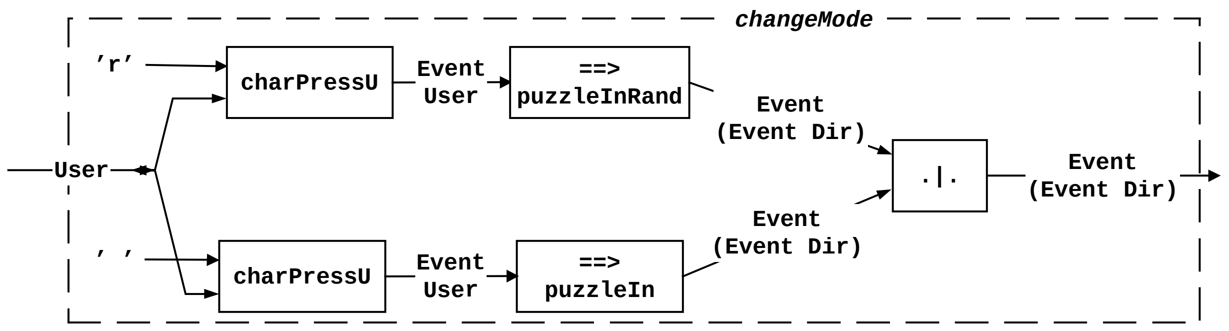

Figure 14. Input mode change

10. Output variations

Another simple variation on the Fifteen Puzzle constructed above is replacing the piece images. We could then redefine `puzzleOut` to have a different single set of piece images wired into its definition, but instead, we generalize it to take the piece image list as an argument, as in Figure 15. (The Fran `overs` function combines a list of images im_1, …, im_n into a single composite image made by stacking all of the images in the list, with im_1 on top and im_n on bottom.)

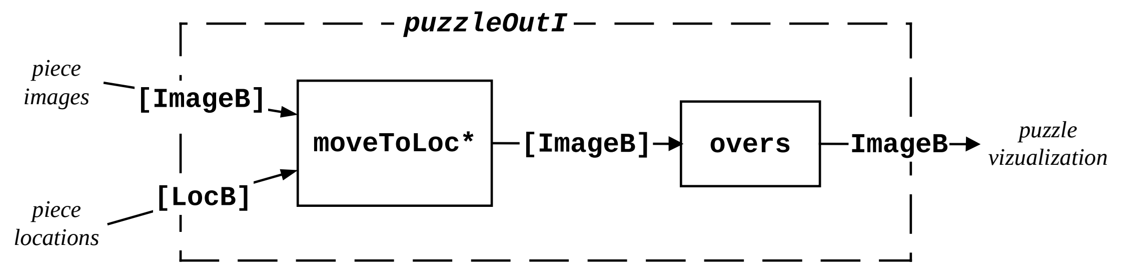

Figure 15. Visualization with given piece images

We generalize `anim` in Figure 16. The original `anim` may be constructed simply by feeding `numIms` into `animI`.

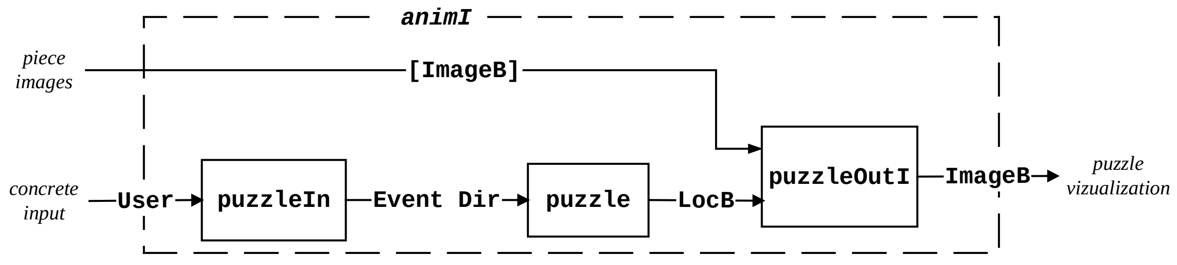

Figure 16. Interactive puzzle with given piece images

Many versions of the puzzle consist of a single picture covering the whole puzzle area. For such a puzzle, we could redefine the piece images, using a function that chops up an image into puzzle-piece-sized parts. This cropping version takes an arbitrary animation (`ImageB` value), which includes static imported pictures as a special case, as shown in Figure 17.

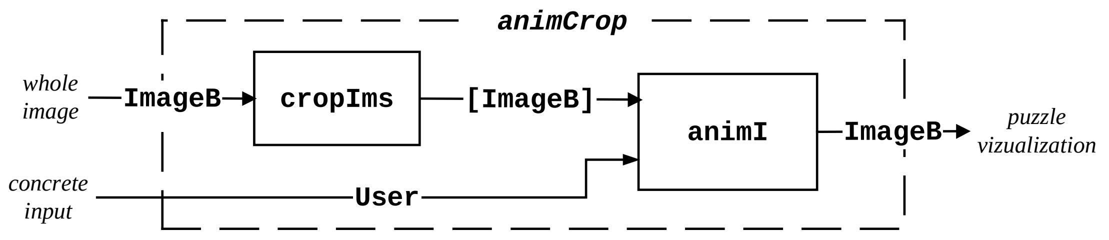

Figure 17. Fifteen Puzzle with chopped-up image

11. Conclusions

In this note, we have developed a Fran implementation of the classic “Fifteen Puzzle”, along with a few variations. Fran’s high-level data types and declarative orientation allowed a *modular* formulation, in which the entire problem into several simpler components. These components are mentally manageable and separately testable.

12. Acknowledgement

Ken Hinckley provided extensive comments on earlier versions of this paper, resulting in greatly improved readability.

References

[1] Conal Elliott, Composing Reactive Animations, *Dr. Dobb's Journal*, July 1998. Expanded version with animated GIFs: `http://www.research.microsoft.com/~conal/fran/tutorial.htm`.

[2] Conal Elliott, Declarative Event-Oriented Programming, MSR-TR-98-24. `ftp://ftp.research.microsoft.-com/pubs`

[3] Conal Elliott and Paul Hudak, Functional Reactive Animation, in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, `http://www.research.microsoft.com/~conal/-papers/icfp97.ps`

[4] Martin Gardner, *The Scientific American book of Mathematical Puzzles and Diversions*, Simon and Schuster, 1959

[5] Paul Hudak and Joseph H. Fasel, A Gentle Introduction to Haskell. *SIGPLAN Notices*, 27(5). See `http://haskell.org/tutorial/index.html` for latest version.

[6] John Hughes, Why Functional Programming Matters. *The Computer Journal*, 32(2), pp. 98-107, April 1989. `http://www.cs.chalmers.se/~rjmh/Papers/whyfp.ps`.

[7] Simon Thompson, *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996. `http://stork.ukc.ac.uk/computer_science/Haskell_craft/`.

Appendix A. Haskell version of fifteen puzzle

User interface vs model

```
puzzleIn  :: User -> Event Dir

data Dir = East | West | North | South deriving Enum

puzzleOut :: [LocB] -> ImageB

-- Locations are (column,row) pairs
type Loc  = (Int, Int)
type LocB = Behavior Loc    -- time-varying

puzzle :: Event Dir -> [LocB]

anim :: User -> ImageB
anim u = puzzleOut (puzzle (puzzleIn u))
```

Model decomposition

```
onePiece :: Event (Dir, Loc) -> Loc -> (Event Loc, LocB)

trackBlank :: [Event Loc] -> LocB

puzzle moveDir = locBs
 where
   (moveFroms, locBs) =
     -- Start each piece going with shared moveWithBlank event and own start
     -- location.  Unzip list of (moveFrom, locB) pairs into pair of lists.
     unzip [ onePiece moveWithBlank start | start <- startLocs ]
   moveWithBlank = moveDir ‘snapshot‘ (trackBlank moveFroms)
```

```
-- Starting locations
blankStart : startLocs =
  [ (col,row) | row <- [0 .. rows-1], col <- [0 .. cols-1] ]

rows, cols :: Num a => a
rows = 4; cols = 4
```

One puzzle piece

```
onePiece moveWBlank startLoc =(moveFrom, locB)
 where
   locB = stepper startLoc moveTo
   (moveFrom, moveTo) =
      unzipE (moveWBlank ‘snapshot‘ locB ‘filterE‘  moveOkay)

   moveOkay ((dir, bLoc), loc)
    | loc +^ dir == bLoc  = Just (loc, bLoc)
    | otherwise           = Nothing

(+^) :: Loc -> Dir -> Loc
(col,row) +^ dir = case dir of
  West  -> (col-1, row  )
  East  -> (col+1, row  )
  North -> (col  , row-1)
  South -> (col  , row+1)
```

Tracking the blank spot

```
trackBlank moveFroms = stepper blankStart (anyE moveFroms)
```

User Interface

```
puzzleIn u = keyPressAny u ‘filterE‘ keyDir

keyDir :: VKey -> Maybe Dir
keyDir vkey = assoc [ (vK_LEFT , West ),
                      (vK_RIGHT, East ),
                      (vK_UP   , North),
                      (vK_DOWN , South) ] vkey

puzzleOut locBs =
  overs [ moveTo (locToPoint locB) im | (locB, im) <- zip locBs numIms ]

numIms :: [ImageB]
numIms = [ stretch 2 (showIm i) | i <- [1 .. rows * cols] ]

-- LocB to Point2B conversion
locToPoint :: LocB -> Point2B
locToPoint locB = point2XY x y
 where
   (col, row) = pairBSplit locB
   row’ = rows - 1 - row
   x = (fromIntB col  + 0.5) * pieceWidth  - puzzleWidth /2
   y = (fromIntB row’ + 0.5) * pieceHeight - puzzleHeight/2
```

```
-- Dimensions
puzzleWidth, puzzleHeight, pieceWidth, pieceHeight   :: RealB
puzzleWidth = 2; puzzleHeight = 2
pieceHeight = puzzleHeight / cols
pieceWidth  = puzzleWidth  / rows

pieceSize :: Vector2B
pieceSize = vector2XY pieceWidth pieceHeight
```

Some variations

```
puzzleInRand, puzzleInSwitch :: User -> Event Dir

-- Random moves
puzzleInRand u =
  updateDone u ‘withElemE_‘ random (0,3) 5 ==> toInt ==> toEnum

-- Switches between random and normal
puzzleInSwitch u =
  switcher (puzzleIn u) (
      charPressU ’r’ u ==> puzzleInRand
  .|. charPressU ’ ’ u ==> puzzleIn    )

-- Crop one image into puzzle-piece images
cropIms :: ImageB -> [ImageB]
cropIms wholeIm =
  -- For each location, crop the whole image
  -- and move to origin
  [ move (origin2 .-. pos) (crop rect wholeIm)
  | loc <- startLocs
  , let pos  = locToPoint (constantB loc)
        rect = rectFromCenterSize
                 pos pieceSize ]

puzzleOutI :: [LocB] -> [ImageB] -> ImageB
puzzleOutI locBs pieceIms =
  overs [ moveTo (locToPoint locB) im | (locB,im) <- zip locBs pieceIms ]

animI :: [ImageB] -> User -> ImageB
animI pieceIms u = puzzleOutI (puzzle (puzzleInSwitch u)) pieceIms
```

Appendix B. Immutable Arrays

The implementation of the Fifteen Puzzle given in this paper suffers from an inherent inefficiency. Every puzzle piece does some work in response to every move request, although in most cases the work is just to see that the piece should do nothing. This property means that the amount of processing required per move is proportional to the puzzle size, and so responsiveness diminishes as the puzzle size grows. Note, however, that for any given move request, there is at most one piece that can respond, namely the one adjacent to the blank space on the side opposite from the requested direction (if that location is on the board). We would like to use this knowledge to “tell” the one relevant piece that it should move, but telling is an imperative notion.

Many other interactive applications have similar properties to the one in this paper. Any time there is an “environment” navigated by behaviors, we will want to direct events to the nearby behaviors (unless they have long-distance perception).

We solve the problem of efficient response to local events by adding two new primitives to Fran. One creates immutable arrays and the other consumes them. The consumer primitive indexes arrays using an index-valued behavior. It applies to arbitrary “generalized behaviors”, such as behaviors, events, `SoundB`, `ImageB`, and `GeometryB`:

```
(!*) :: (GBehavior bv, Ix a) => Array a bv -> Behavior a -> bv
```

Think of "`!*`" as an optimized version of some nested conditionals:

```
ifB (ix ==* constantB ix0) (arr ! ix0) $
...
ifB (ix ==* constantB ixn) (arr ! ixn) $
error "bad index"
```

In addition to the existing Haskell operations for creating arrays, a new primitive builds arrays of events:

```
arrayE :: Ix a => (a,a) -> Event (a,b) -> Array a (Event b)
```

Compare with the basic array constructor:

```
array :: Ix a => (a,a) -> [(a,b)] -> Array a b
```

In practice, `arrayE` is a way to "post" an occurrence of a run-time-selected event. For instance, we might have a large array of component animations `bv_1`, …, `bv_n`, based on `b`-valued events `e_1`, ..., `e_n`. If we construct the array of events using arrayE, then an `(Int,b)`-valued event can choose which of the `bv_i` should react.

Given these new primitives, we can construct a new implementation of the Fifteen Puzzle that not only eliminates the inherent inefficiency described above, but is also a higher level specification. The idea is that the puzzle routes "`moveTo`" events to the puzzle locations using `arrayE`, and the pieces tune in to just the `moveTo` events of interest using "`!*`".

Here is the definition. It replaces `puzzle`, `onePiece`, and `trackBlank`, and uses "`-^`", an analog to "`+^`".

```
puzzle :: Event Dir -> [LocB]
puzzle moveDir = map pieceLoc loc0s
 where
   moveFromTo :: Event (Loc,Loc)
   moveFromTo = moveDir ‘snapshot‘ blankLoc
                      ==>        \ (dir,to) -> (to-^dir,to)
                     ‘suchThat‘ (legalLoc . fst)

   blankLoc :: LocB
   blankLoc = stepper blankLoc0 (moveFromTo ==> fst)

   moveTos :: Array Loc (Event Loc)
   moveTos = arrayE puzzleBounds moveFromTo

   pieceLoc :: Loc -> LocB
   pieceLoc loc0 = locB
    where
      locB = stepper loc0 (moveTos !* locB)

legalLoc :: Loc -> Bool
legalLoc (col,row) = 0 <= col && col < cols && 0 <= row && row < rows
```