Two-handed Image Navigation in Fran

Conal Elliott

October, 1998

Technical Report
MSR-TR-98-56

Presented at the Glasgow Functional Programming Group Workshop
in Pitlochry, Scotland, September 1998

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA  98052
1. Introduction

This paper describes a program for allowing a user to navigate smoothly in a 2D (possibly animated) image, performing pan, zoom and rotation. The program is written in Haskell [5][6], using the Fran interactive animation library [3][2][1]. The program is directly inspired from an application by Ken Hinckley [4] written in C++, using OpenGL.

The idea of image navigation is to allow a user to explore a complex image such as a map, moving smoothly between different locations, scales, and orientations. Single-handed input (e.g., with a single mouse) requires the user to change “input modes”, interrupting the flow of the user’s main task. Two-handed input eliminates the need for conscious mode changes, freeing the user to concentrate on the navigation task.

Imagine that the image being explored is printed on a uniformly stretchable material, sticky on the top and smooth on the bottom, lying on a tabletop. When a hand touches the material at some location, whatever location on the material is in contact with the hand becomes stuck to it. While only a single hand is touching the material, the material is only allowed to pan (translate), but not rotate or zoom. While both hands are touching, the material is allowed to pan, rotate, and zoom simultaneously. Figure 1 shows snapshots of a star-shaped image being moved around with the right hand, while Figure 2 shows the two-handed pan/zoom/rotate behavior.¹

---

¹ Scan the frames from left to right, starting with the first row. For running, color animations of these two examples, see ftp://ftp.research.microsoft.com/pub/tr/tr-98-56/animations.htm.
See the Fran references for details about Fran. We give here only a very brief overview. The two main concepts are behaviors and events, both polymorphic. A behavior is function of continuous time, or in other words, a “time-varying value”. As a convention, names like “BoolB” are synonyms for types like “Behavior Bool”. An event is a stream of occurrences, each of which is a time/value pair. Another type, User, contains a collection of behaviors and events describing user input. We explain some of the functions used in the code below, omitting the ones whose meanings are apparent from their names.

2. Modeling Hands

We represent a “hand” as something that has a position and can indicate grasping and releasing events. It will also be convenient for a hand to contain a boolean behavior saying whether the hand is currently grasping:

```
-- Which hand, motion, grasp, release and grasping
data Hand = Hand Point2B (Event ()) (Event ()) BoolB
```

This convenience function mkHand computes the grasping boolean, while debouncing its given grasp and release events, which is particularly useful for keypress-based hands.2

```
mkHand :: Point2B -> Event () -> Event () -> Hand
mkHand pos tryGrasp tryRelease = Hand pos grasp release grasping
    where
        grasping = stepper False (grasp -=> True .|. release -=> False)
        grasp    = tryGrasp `whenE` notB grasping
        release  = tryRelease 'whenE' grasping
```

We will want to render hands for user feedback. For precision, show a small circle at the exact point of grasping, colored green while grasping and red while not.3

```
data WhichHand = LeftH | RightH

renderHand :: WhichHand -> Hand -> ImageB
renderHand lr (Hand pos _ _ grasping) =
    moveTo pos (stretch 0.02 (withColor (ifB grasping green red) circle) 'over'
                ifB grasping (importHand lr True )
                (importHand lr False))
```

Left vs. right hands use different base images, as do grasping vs. relaxed. We use four image files:

```
data WhichHand = LeftH | RightH

handName :: WhichHand -> Bool -> String
handName LeftH  True  = "hand_pinch_l_32.bmp"
handName LeftH  False = "hand_relax_l_32.bmp"
handName RightH True  = "hand_pinch_r_32.bmp"
handName RightH False = "hand_relax_r_32.bmp"
```

2 This definition uses the following Fran functions (with “-=>” binding more tightly than “.|. ”):

```
stepper :: a -> Event a -> Behavior a     -- Generates a piecewise-constant behavior, with given start
         --  value, changing on each occurrence of given event
(-=>)  :: Event a -> b -> Event b         -- Replace event data with constant value
(.|.)  :: Event a -> Event a -> Event a   -- Merge two events into a single one
whenE  :: Event a -> BoolB -> Event a     -- Keep occurrences when predicate true
```

3 The function ifB is a time-lifted version of the if-then-else operator. It applies to “generalized behaviors”, i.e. members of the GBehavior type class, which contains all behaviors and events, as well as ImageB and User.

```
ifB :: GBehavior bv -> BoolB -> bv -> bv -> bv
```
The imported hands come from importing the appropriate bitmap and translating so that the grasp point is at the origin.

\[
\text{importHand} \equiv \lambda \text{WhichHand} . \to \text{Bool} . \to \text{ImageB} \\
\text{importHand} \ \text{lr} \ \text{grasping} = \text{move} \ (\text{origin2} \ .-. \ \text{pinchPos}) \ \text{imB} \\
\text{where} \\
\text{imB} = \text{importBitmap} \ (\text{handName} \ \text{lr} \ \text{grasping}) \\
\text{pinchPos} = \text{point2XY} \ (\text{lrFlip} \ \text{lr} \ \text{fromImportPixel} \ 14) \ \text{fromImportPixel} \ 9 \\
\text{lrFlip} \ \text{LeftH} = \text{id} \\
\text{lrFlip} \ \text{RightH} = \text{negate}
\]

With these basic tools, we now define some hand generators.

\[
\text{type HandGen} = \text{User} \to \text{Hand} \\
\text{mouseHand, stylusHand, keyHand, stylusOrKeyHand} : : \text{HandGen}
\]

Stylus and mouse hands are simple:

\[
\text{stylusHand} \ \text{u} = \text{mkHand} \ (\text{stylus} \ \text{u}) \ (\text{stylusDown} \ \text{u}) \ (\text{stylusUp} \ \text{u}) \\
\text{mouseHand} \ \text{u} = \text{mkHand} \ (\text{mouse} \ \text{u}) \ (\text{lbp} \ \text{u}) \ (\text{lbr} \ \text{u})
\]

For key-based hands, use the shift key to grasp and release. The arrow keys indicate direction of movement. The direction is actually a velocity vector, with the left and right keys contributing -1 and 1 in X, and the down and up keys contributing -1 and 1 in Y.

\[
\text{keyHand} \ \text{u} = \text{mkHand} \ \text{pos} \ (\text{keyPress} \ \text{vK\_SHIFT} \ \text{u}) \ (\text{keyRelease} \ \text{vK\_SHIFT} \ \text{u}) \\
\text{where} \\
\text{pos} = \text{origin2} \ .+. \ \text{atRate} \ \text{vel} \ \text{u} \\
\text{vel} = \text{vector2XY} \ (\text{keyVal} \ \text{vK\_LEFT} \ \text{vK\_RIGHT} \ \text{u}) \ (\text{keyVal} \ \text{vK\_DOWN} \ \text{vK\_UP} \ \text{u}) \\
\text{keyVal} :: \text{VKey} \to \text{VKey} \to \text{User} \to \text{RealB} \\
\text{keyVal} \ \text{neg} \ \text{pos} \ \text{u} = \text{dirVal} \ \text{pos} - \text{dirVal} \ \text{neg} \\
\text{where} \\
\text{dirVal} \ \text{key} = \text{stepper} \ 0 \ (\text{keyPress} \ \text{key} \ \text{u} -\to 1 \ .|. \ \text{keyRelease} \ \text{key} \ \text{u} -\to 0)
\]

Finally, define a hand generator that uses the stylus if present and otherwise uses the keyboard.

\[
\text{stylusOrKeyHand} \ \text{u} | \text{stylusPresent} \ \text{u} = \text{stylusHand} \ \text{u} \\
\text{otherwise} = \text{keyHand} \ \text{u}
\]

3. Navigation

The heart of our application is the navigate function, which maps two hands to a time-varying 2D transform:

\[
\text{navigate} :: \text{Hand} \to \text{Hand} \to \text{Transform2B} \\
\text{navigate} \ (\text{Hand} \ \text{posA} \ \text{graspA} \ \text{releaseA} \ \text{graspingA}) \\
(\text{Hand} \ \text{posB} \ \text{graspB} \ \text{releaseB} \ \text{graspingB}) = \text{xfB} \\
\text{where} ...
\]

The transform xFB is composed from two pieces: atChange, which captures what the transform was at the last mode change, and sinceChange, which is the relative transform since the last mode change.\(^\text{4}\)

\[
\text{xfB, atChange, sinceChange} :: \text{Transform2B} \\
\text{xfB} = \text{sinceChange} \ \text{compose2} \ \text{atChange} \\
\text{atChange} = \text{stepper} \ \text{S.identity2} \ (\text{change} \ \text{snapshot\_}) \ \text{xfB} \\
\text{change} = \text{graspA} \ .|. \ \text{releaseA} \ .|. \ \text{graspB} \ .|. \ \text{releaseB}
\]

\[^\text{4}\] The definitions in the remainder of this section are all in the scope of navigate’s “where” clause. The constant S.identity2 is the 2D static (not time-varying) identity transform. The \text{snapshot\_} function replaces an event occurrence’s value with a snapshot of a behavior at the occurrence time.
There are four modes: pan/zoom/rotate, pan with hand A or B, or stationary.

```haskell
sinceChange = ifB graspingA
               (ifB graspingB pzp panA)
               (ifB graspingB pzn panB identity2)
```

The following function helps define the panning and zooming behaviors. Given a hand position, it captures the position at last mode change and a relative panning transform.

```haskell
startAndPan :: Point2B -> (Point2B, Transform2B)
startAndPan pos = (pos0, pan)
  where
    pos0 = stepper undefined (change 'snapshot_' pos)
    pan  = translate2 (pos .-. pos0)
```

Then apply `startAndPan` to the two hands:

```haskell
posA0, posB0 :: Point2B
(posA0, panA) = startAndPan posA
(posB0, panB) = startAndPan posB
```

In pan/zoom/rotate mode, we synthesize a transform \( X \), such that \( X \ a \ 0 = a \) and \( X \ b \ 0 = b \), for time-varying points \( a \) and \( b \) and their values \( a \ 0 \) and \( b \ 0 \) at the last mode change. First, decompose \( X \) into translation and linear components:

\[
X = \text{translate} \ d \circ W
\]

Consider the difference vectors, \( v = a - b \) and \( v_0 = a_0 - b_0 \):

\[
v = b - a = X \ b \ 0 - X \ a \ 0 = (W \ b \ 0 + d) - (W \ a \ 0 + d) = W \ b \ 0 - W \ a \ 0 = W (b \ 0 - a \ 0) \text{ (by linearity of } W) = W v_0
\]

Consider \( v_0 \) and \( v \) in polar form:

\[
v_0 = \text{vector2polar} \ r_0 \ \theta_0
\]

\[
v = \text{vector2polar} \ r \ \theta
\]

In mapping \( v_0 \) to \( v \), \( W \) must stretch \( v_0 \) to \( v \)’s length, and rotate \( v_0 \) to \( v \)’s orientation, so we have

\[
W = \text{scale} \ (r/r_0) \circ \text{rotate} \ (\theta - \theta_0)
\]

The translation vector \( d \) then makes up the difference between \( a \) and \( W a_0 \) (or between \( b \) and \( W b_0 \)), i.e., \( a = X a_0 = W a_0 + d \), so

\[
d = a - W a_0
\]

The implementation of \( \text{pzr} \) then follows:

```haskell
pzr = translate2 d `compose2` w
where
  w = uscale2 (r / r0) `compose2` rotate2 (theta - theta0)
  d = posA .-. w *% posA0

  (r0, theta0) = vector2PolarCoords (posB0 .-. posA0)
  (r, theta) = vector2PolarCoords (posB .-. posA )
```

---

5 The “*%” operator applies a 2D transform to a point, vector, or image animation.

```haskell
(*%) :: Transformable2B a -> Transform2B -> a -> a
```
4. Putting it Together

There is not much more to do. The function \texttt{showNav} takes two hand generators, an image to explore, and user input. It produces an animation consisting of renderings of the two hands generated on top of the given image transformed by \texttt{navigate}.

\begin{verbatim}
showNav :: HandGen -> HandGen -> ImageB -> User -> ImageB
showNav genA genB imb u = renderHand RightH handA 'over'
  renderHand LeftH  handB 'over'
  navigate handA handB *% imb

where
  handA = genA u
  handB = genB u
\end{verbatim}

Our main program simply gives the two hand generators and a star image to \texttt{showNav}, and displays the result.

\begin{verbatim}
main = displayU (showNav mouseHand stylusOrKeyHand (withColor yellow (star 3 7)))
\end{verbatim}

5. Conclusions

In this paper, we have presented a \textit{complete} implementation of two-handed image navigation, using Fran. While the C++ program that inspired this exercise is quite long and complex, the Fran version is short, modular and we feel captures the essence of the behavior being modeled.

Fran is freely available at \url{http://research.microsoft.com/~conal/Fran}.

References


