

# Motion: a motion library in Haskell

Kyle Ross <kyle@osl.iu.edu>

this is KVS version-2

## 1 Introduction

The purpose of *Motion* is to allow users of OpenGL [6] in Haskell [1] to easily script interpolations and motion, avoiding the need to manually pass state information and allowing rapid development with minimal coding.

Interpolations can be used in a variety of ways from drawing Bézier curves to moving a camera along a Catmull Rom interpolated path to generating textures to performing a quaternion-interpolated rotation around an object—the notion of interpolation encoded in the library’s classes is general enough to allow use in many contexts and in service of varied goals.

Since *Motion* is a very general library, it is impossible to define what, specifically, it does. However, this document will describe the components of the library and will walk through the interesting bits of the demonstration codes included with the library.

This document will assume knowledge of Haskell (especially of the class system [2, 5]) and of HOpenGL [4], the Haskell binding for OpenGL. If the reader is familiar with OpenGL, HOpenGL should be fairly simple to grasp.

Because of assorted technical issues, HOpenGL and *Motion* require to use the Glasgow Haskell Compiler [3]. The “sub-library” Quaternion will be described only in the associated example. (See Section 3.5.)

## 2 Components of *Motion*

### 2.1 Interp a: a general interpolation

The class `Interp a` encodes the most fundamental idea of *Motion*—an interpolation.

```
class Interp a where  
  interp  :: Double → a → a → a
```

```
interpF :: a -> a -> Double -> a
interpF x y t = interp t x y
```

The class requires a basic definition (viz., `interp`) of what it means to interpolate values of type `a` and defines a convenience function `interpF` that simply switches argument order in the default implementation.

Now, we can declare, for instance, that `Double` is interpolatable with the following instance declaration:

```
instance Interp Double where
  interp t x y = x + (y - x) * t
```

This is the standard definition of a linear interpolation. We can test that this works:<sup>1</sup>

```
Prelude Motion> interp 0 0 (5 :: Double)
0.0
Prelude Motion> interp 0.5 0 (5 :: Double)
2.5
Prelude Motion> interp 1 0 (5 :: Double)
5.0
```

`Motion` provides `Interp` instance declarations for many common built-in types; of course, users can add additional declarations for arbitrary types. `newtype` declarations can be used to give multiple interpolation interpretations to the same type (i.e., to define multiple ways to interpolate a single type).

## 2.2 InterpStream a: a stream of interpolated values

While `Interp` gives us a very basic notion of interpolation, since we are concerned with animation and motion, what we would really like is a way to generate *streams* of interpolations—given suitable parameters, we would like to be able to represent notions of “move from here to there in this way”. To accomplish this, we define `InterpStream`, representing such a stream:

```
type InterpStream a = Base -> [a]

type Action          = IO ()
type ActionStream = InterpStream Action
```

But, what is `Base`? Well, we must specify how many steps the interpolation should have and what values the interpolation parameter should move through. Since we have chosen `Double` as the type of the interpolation parameter, we can define `Base` as:

```
type Base = [Double]
```

The ways of constructing bases and the operations for transforming them are discussed in Section 2.4.

---

<sup>1</sup>We must add `Double` type annotations because Haskell has polymorphic numeric constants.

### 2.2.1 InterpStream Manipulations

We can now define some combinators over `InterpStreams`—manipulations on `InterpStream`. In this document, we will focus on giving some intuition as to the operation of the combinators rather than explicating their definitions; interested users are encouraged see the source of `Motion` for more details. Haskell programmers have a strong tradition of creating lovely and unpronounceable operators; we will not deviate from this heritage.

**The pairing combinator**, `x ++ y`, combines two `InterpStreams` into a stream pairing associated values from `x` and `y`; of course, this assumes that they have equal lengths, which will always be the case if they are fed the same `Base`.

**The transformation combinator**, `x >>> y`, applies a function `f` pointwise to the stream `x`. As an example, we can take a stream of numbers between 0 and 10 and print them out:

```
(linear 0 10) >>> print
```

**The repetition combinator**, `n *** x`, repeats an interpolation `x`, `n` times. We can use this to build simple loops. (Additionally, the `****` combinator builds dropped-frame loops, which can be useful if the start and end points are equal to avoid a duplicated frame.) (There are also infinite-loop combinators, `inf` and `indD`, providing infinite versions of `***` and `****`, respectively.)

**The sequencing combinator**, `x >>>> y`, splits the basis in half, “riffle-wise”, passing the first half to `x`, then passing the second to `y`.

**The “rock” combinator**, `(>><<) x` is a specialisation of `>>>>`. It does a “from here to there and back” interpolation, running the stream `x` forward then backward (much like rocking the jog dial back and forth when editing video; hence the name).

**The reverse combinator**, `rev` reverses a stream. For example,

```
rev (linear 0 5)
```

interpolates from 5 to 0.

## 2.3 Interpolation Generators

Now that we know what it means to be “interpolatable” and how to manipulate interpolations, how do we *create* an interpolation? *Motion* provides several ways to accomplish this.

**A constant interpolation** along a single value “lifts” a value to an interpolation; this is the simplest possible interpolation and is implemented in *Motion* as `constant x` for any value `x`.

**A linear interpolation** takes two values and interpolates between them according to a `Base` (applied later). For example, `linear 0 5` interpolates between 0 and 5.

**A bicubic Bézier interpolation**, `bezier`,<sup>2</sup> takes four control points and applies the de Casteljau spline construction using the identity function at each iteration.

**Bicubic Catmull-Rom and B-Spline interpolations**, `catmullRom` and `bSpline`, respectively, are also provided. *Motion* only provides 4-control-point versions, but these can be combined easily to extend these to “sliding-window” curve interpolations. Only cubic interpolations are defined, but it would be easy to define other orders of interpolation (Perhaps the order could be Peano-encoded into a type to have the class system automatically select or generate the proper function.).

## 2.4 Base Manipulations

Bases can be constructed with the functions `baseWith n` (creates a unit basis with `n` steps) and `baseBy s` (creates a unit basis stepped by `s`).

Bases can be manipulated with the functions `baseTranslate x` (adds `x` to each element of the base), `baseScale x` (multiplies each element of the base by `x`), `baseRotate s` (rotates the base by `s` steps in “circular-list fashion”), and `baseDownSample s` (selects every `s`<sup>th</sup> element from the base).

These are all implemented in the obvious ways.

## 2.5 Time: running an interpolation

So far, we have discussed interpolations abstractly—without any notion of how the interpolation parameter is being varied. The class `Time` allows to define

---

<sup>2</sup>The name of the function is actually “bézier”, but cannot be typeset this way here for technical reasons.

various “interpolation semantics”. We define a `Sequence`—a sequence of actions interpolated according to a specific notion of time. A `Sequence` can be created by tying an `ActionStream` (i.e., an `InterpStream (IO ())`) to a `Time` interpolation. This is done using the `x && t` combinator, which “ties” an `ActionStream` `x` to a `Time` interpolation `t`. For example:

```
((linear 0 5) >>> print) && instantaneous
```

creates a `Sequence` that instantaneously prints the interpolated values between 0 and 5.

Now, we must commit to a particular base to determine the values over which the interpolation parameter will range. We can do this with the `b ==> s` combinator, which “finalises” a `Sequence` `s` with a `Base` `b`. For example (our first complete example!):

```
(baseWith 10)
  ==>
  (((linear (0 :: Double) 5) >>> print) &&
instantaneous)
```

prints the output:

```
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
```

Other `Times` define interpolation across time and callback generators.

**The parallelisation combinator**, `s1 ++ s2`, creates an `ActionStream` whose actions consist of an `Action` from `s1` and an `Action` from `s2`. `s1`’s actions happen *before* `s2`’s actions at each step.

### 3 Example Mini-Applications

This section briefly describes the demonstration codes distributed with `Motion`. For each demo program, we provide three paragraphs’ explanation, detailing the *what* (What you should expect to see when running the demo.), *why* (What features of `Motion` the demo shows.), and *how* (What the interesting bits of the implementation do.).

### 3.1 Simple.hs

**What:** Prints the sequence of squares of 0 through 10 (0, 1, 4, ..., 100).

**Why:** This is a demonstration of the simplest use of *Motion*, involving none of the complications involved in using HOpenGL, etc.

**How:** This is very similar to the example seen earlier.

```
(baseWith 10)                — do 10 steps
  =>>>=
    (((linear (0 :: Integer) 10) — between 0 and 10
      >> (^ 2)
— square each number
      >> print)                — and print it
    &&&
      instantaneous)           — do it all NOW!
```

### 3.2 BezCurve.hs

**What:** Draws a Bézier curve, a Catmull-Rom curve, and a B-Spline curve interpolating between the same four control points each time.

**Why:** This demonstrates that *Motion* can replace the operation of OpenGL's curve "evaluators" and shows a very simple program using both *Motion* and HOpenGL.

**How:** We make use of the `++` combinator to simultaneously perform the three interpolations. `tie` combines the results so that the stream can be tied with `&&&`; we could as well have used `++=++` instead with three "copies" of `instantaneous`.

```
baseBy 0.01
  =>>>=
    (((((( bezier c1 c2 c3 c4)
      ++ (bSpline c1 c2 c3 c4))
      ++ (catmullRom c1 c2 c3 c4))
    >>
      (tie . pair (tie . both rp) rp))
    &&& instantaneous)
  where rp = renderPrimitive Points . vertex
        c1 = ((Vertex3 (-4) (-4) 0) :: Vertex3 Float)
        c2 = Vertex3 (-2) 4 0
        c3 = Vertex3 2 (-4) 0
        c4 = Vertex3 4 4 0
```

### 3.3 BezCurveTime.hs

**What:** Draws a Bézier curve, a Catmull-Rom curve, and a B-Spline curve interpolating between the same four control points each time; draws the associated points on each curve (i.e., those that have the same value for the interpolation parameter) at once, pausing between displaying each triple of points.

**Why:** This demonstrates a *very* simple proof-of-concept animation. N.b., this program has *very* poor OpenGL style—it manually calls the update / flush functions. This was done to simplify the Time definition for beginners.

**How:** We implement a “fake time” Time sequence that manually iterates through its “frames”. This *should never be used in a real application*, as it violates almost every style directive of OpenGL; it is for demonstrative purposes only. At each step, it draws an interpolated point on each curve, this time in a different colour for each curve.

```
data FakeTime = FakeTime
```

```
instance Time FakeTime where
  base =>>>= (Sequence s) =
    runAll $ s base
    where runAll [] = return ()
          runAll [(a, t)] = do a
          runAll ((a, t) : (b, u) : xs) =
            do a
              flush
              addTimerCallback delay (runAll ((b, u) : xs))

          delay = 200
```

```
baseWith 20
```

```
=>>>=
  (((((bzier c1 c2 c3 c4)
    ++ (bSpline c1 c2 c3 c4))
    ++ (catmullRom c1 c2 c3 c4))
  >>
    (tie . pair (tie . pair rpRed rpYel) rpBlu))
  &&& (constant FakeTime))
  where rp = renderPrimitive Points . vertex
        rpc c v = do color c
                  rp v
        rpRed = rpc (Color3 (1 :: GLfloat) 0 0)
        rpYel = rpc (Color3 (1 :: GLfloat) 1 0)
        rpBlu = rpc (Color3 (0 :: GLfloat) 0 1)
```

```

c1 = ((Vertex3 (-4) (-4) 0) :: Vertex3 Float)
c2 = Vertex3 (-2) 4 0
c3 = Vertex3 2 (-4) 0
c4 = Vertex3 4 4 0

```

### 3.4 Scene.hs

**What:** Displays a simple 3d scene rotating whilst the camera zooms in and out.

**Why:** This shows a better integration between `Motion` and `HOpenGL`, demonstrating use of the `Callback` interpolation, whose details are a bit beyond the scope of this document. (Essentially, though, a `Callback` interpolation creates a callback function that responds to `Execute` (draw the frame) and `Advance` (move to the next frame) directives.) Also, it demonstrates use of `infD` and `(>><<)` to create a looped animation, using `+=+` to combine the rotation with the zoom.

**How:** `displayInt` draws a frame and `setView` sets up the camera. What is nice about using `Motion` is that these use *no global variables*. How is this possible? `Motion` uses a single hidden global variable to maintain state, performs updates “in the background”, and passes the proper values to the functions when each frame is drawn.

```

autoMkCallbackT $ \cb →
  (baseWith 30)
    =>>>=
      (infD (((linear (0 :: GLfloat) 360)
              >> displayInt)
            +=+
              ((>><<) ((linear (45 :: GLdouble) 90)
                          >> setView)))
            && ((constant cb))))

```

### 3.5 SceneQ.hs

**What:** Displays a set of smooth rotations around a simple model in 3d.

**Why:** This displays the use of a sliding-window B-spline quaternion motion interpolation.

**How:** Using the quaternion “sub-library”, we define `Slerp`’s on quaternions. To be able to read the following definition, it should suffice to know that `x <*> q`

is the product of a scalar  $x$  with a quaternion  $q$ ,  $\langle + \rangle$  is the sum of two quaternions,  $\langle . \rangle$  is the dot-product of two quaternions, and `norm` normalises a quaternion.

```
newtype Quaternion = Quat (Double, Double, Double, Double)
```

```
instance Interp Quaternion where
  interp = slerp
```

```
slerp u q1 q2 = (a <*> q1') <+> (b <*> q2')
               where t   = acos $ q1' <.> q2'
                     st  = sin t
                     a   = (sin $ (1 - u) * t) / st
                     b   = (sin $ u * t) / st
                     q1' = norm q1
                     q2' = norm q2
```

Now, using a straightforward sliding-window B-spline interpolation, we do a rotation interpolation, passing the display function (`displayInt`) the interpolated quaternion for each frame:

```
(baseWith 100)
  =>>>=
    ((buildSlidingWindow [a, b, c, d, e, f, g] displayInt)
     && ((constant cb)))
  where a = Quat (1, 0, 0, 0)
        b = Quat (halfPi, 0, 0, 1)
        c = Quat (halfPi, 1, 0, 0)
        d = Quat (halfPi, 0, 1, 0)
        e = Quat (halfPi, 1, 1, 0)
        f = Quat (halfPi, 0, 1, 1)
        g = Quat (halfPi, 1, 0, 1)
        halfPi = 3.14159 / 2
```

To finish things, we show the function that uses a quaternion to perform a rotation:

```
qRotate (Quat (w, x, y, z)) =
  rotate (degOfRad w) (Vector3 x y z)
```

### 3.6 SceneInteraction.hs

**What:** Displays a simple 3d scene rotating whilst the camera zooms in and out; allows to click and drag the mouse in X to change the background colour.

**Why:** Until now, all our examples with *Motion* have been *non-interactive*. As a proof-of-concept—to show that there is every possibility to combine use of *Motion* with interaction—we provide this simple example.

**How:** To do interaction, we cannot allow `Motion` to handle the global variables involved in the interactive portion of the program (because the values of such variables do *not* follow a predetermined interpolation!); these must be handled in the “normal” way they would be in a C program. The interpolation for the motion of the scene and zooming of the camera are performed exactly as in `Scene.hs`.

### 3.7 Texture.hs

**What:** Displays a teapot spinning in Z then Y then X (and then repeating) with interpolated texturing.

**Why:** This demonstrates an alternative use of an interpolation—besides interpolating the motion of the pot, the texture is also generated, using a Bézier interpolation between 4 colours.

**How:** To generate rotation in different dimensions, we simply tag the rotation amount by a direction and define interpolations on the tagged rotations. To sequence the rotations with uniform period (i.e., so each rotation completes in the same amount of real time), we must downsample the Z rotation (since  $\ggg$  uniformly downsamples between its two arguments, the Z rotation would otherwise take as long as the *sum* of the X and Y rotations.).

```
autoMkCallbackT $ \cb →
(baseWith 180)
  =>>>=
  (infD
    (((((linear (ZRot 0) (ZRot 360))
      >> (displayInt m)) . baseDownSample 2)
    >>>
      (((linear (YRot 0) (YRot 360))
        >> (displayInt m))
      >>>
        ((linear (XRot 0) (XRot 360))
          >> (displayInt m))))))
    && ((constant cb))))
```

To generate the texture, we interpolate along a Bézier curve between four colour points. We interpolate between two colours by numerically interpreting their components.<sup>3</sup>

```
instance Interp a => Interp (Color4 a) where
  interp t (Color4 r1 g1 b1 a1) (Color4 r2 g2 b2 a2) =
    Color4 (interp t r1 r2)
```

---

<sup>3</sup>We must, most unfortunately, refer directly to the implementation-internal type `GHC.Word.Word8` because Haskell does not allow to use a type synonym in a class instance declaration.

```

        (interp t g1 g2)
        (interp t b1 b2)
        (interp t a1 a2)

instance Interp GHC.Word.Word8 where
  interp t x y =
    round $ (fromIntegral x) * t
      + (fromIntegral y) * (1 - t)

withStripeImage act =
  withArray (( >>> )) (bezier (Color4 255 0 0 255)
                              (Color4 255 255 0 255)
                              (Color4 0 255 0 255)
                              (Color4 0 0 255 255))
    (baseWith $ fromIntegral w)
    $ act . PixelData RGBA UnsignedByte
where TextureSize1D w = stripeImageWidth

```

## References

- [1] *Haskell 98 Language and Libraries: The Revised Report*, December 2002. <http://www.haskell.org/onlinereport/index.html>.
- [2] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, New York, NY, USA, June 1995. ACM Press.
- [3] S. Marlow, S. Peyton-Jones, et al. The glasgow haskell compiler. <http://www.haskell.org/ghc/>.
- [4] S. Panne. Hopengl. <http://www.haskell.org/HOpenGL/>.
- [5] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.
- [6] M. Woo, Davis, and M. B. Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.