Hello

Table of Contents

About

Welcome to my homepage. Here you can find my devlog.

I think browser engines are too complex. So, this website is a PDF document.

There is an RSS feed at ./rss.xml.

Projects

litmark/

: a portable literate-programming tangler written in Fennel

mill.py/

: a Markdown interface to LLMs written in Python

Devlog

2026-06-15 strokeWeight Workaround

#8847: Stroke Scaling is Different in p5 WebGL and WebGPU

This bug means that the scale() transformation is not applied to strokeWeight(). So, if you want to apply a scaling factor to stroke thickness, then you need to explicitly multiply the thickness by the scale each time you call strokeWeight().

Imagine that you want to apply a scaling factor to a group of drawing calls, and this group has some indirect calls to strokeWeight(). Now, you have to start adding a scaling factor to these calls.

To make things worse, scale() compounds. In the code snippet below, the third circle is six times larger than the first circle.

```
draw = do
  circle { x = 0, y = 0, r = 10 }
  scale 2
  circle { x = 0, y = 0, r = 10 }
  scale 3
  circle { x = 0, y = 0, r = 10 }
```

If you combine this effect with indirect calls to `scale()`, you're looking at an increasingly hairy situation.

And. It. Gets. Worse. :)

The effect of `scale()` calls can be contained by calling `push()` and `pop()` before and after. So, you can even have something such as below. Here, the third circle is the same size as before, and the fourth circle is the same size as the second circle. Yes, indirect calls to `scale()` would be contained as well.

```
draw = do
  circle { x = 0, y = 0, r = 10 }
  scale 2
  circle { x = 0, y = 0, r = 10 }
  push
  scale 3
  circle { x = 0, y = 0, r = 10 }
  pop
  circle { x = 0, y = 0, r = 10 }
```

So, what if you want to have a function that represents a figure that happens to have some strokes with a particular stroke thickness? You may want to use this function in a context where you need to scale the figure.

It turns out this problem is not that hard to work around with Effectful. When we encounter `scale()`, `push()`, or `pop()`, we can store the current scale in a `State effect`. Then, `strokeWeight()` can simply read from the state. This is how that code currently looks in my ``engine``:

```
-- Group calls to push and pop for
-- ergonomics
group :: State Scale :> es
      => Eff es () -> Eff es ()
group eff = do
  Scale ss@(s :| ss') <- get @Scale
  put $ Scale $ s :| s : ss'
  push
  eff
  pop
  put $ Scale ss

scale :: State Scale :> es
      => Double -> Eff es ()
scale f = do
  Scale (s :| ss) <- get @Scale
  put $ Scale $ (s * f) :| ss
  liftIO $ Foreign.scale f

strokeWeight :: State Scale :> es
             => Double -> Eff es ()
strokeWeight v = do
  Scale (s :| _) <- get @Scale
  liftIO $ Foreign.strokeWeight $ s * v
```

2026-06-14 Football Field

Hello, world cup!

62

Figure 1. Football field

2026-06-10 GHC WASM Example

https://tildegit.org/unworriedsafari/ghc-wasm-hello

2026-06-10 Reactive Banana Integration

I'm integrating Effectful with Reactive Banana. It doesn't appear to be super straightforward with Reactive Banana's MomentIO monad. The following code works at least.

```
withEffToIO (ConcUnlift Ephemeral
              Unlimited) $ \runInIO -> do
  network <- compile $ do
    eDraw <- fromAddHandler drawAddHandler
    eDeltaTime <- mapEventIO
                  (\_ -> (*1e-3) <$>
                         runInIO deltaTime)
                  eDraw
    bGameState <- accumB gs $ pure stepTime
                              <@> eDeltaTime
    join . liftIO . runInIO $
      Devtools.sceneMomentIO d
        fireGameState fireSceneSwitch
        bGameState
```

2026-06-07 Reader Effect

Removed the p-parameter by using the Reader effect. Also, eliminated the use of Maybe by putting the IO action itself in the state.

```
evalState (return () :: IO ()) $ do
  (drawAddHandler, fireDraw) <-
    liftIO newAddHandler

  setup $ do
    h <- windowHeight
    w <- windowWidth
    createWebGLCanvas w h

    font  <- loadFont "/static/monogram.ttf"
    sheet <- loadImage "/static/Cat_Grey.png"
    put $ fireDraw (font, sheet)

  draw $ get >>= liftIO
```

2026-06-07 Effectful Example

Another example of Effectful, distributing a single state over multiple JS callbacks.

```
evalState (Nothing :: Maybe Assets) $ do
  setup p $ do
    h <- windowHeight p
    w <- windowWidth p
    createWebGLCanvas p w h
    font  <- loadFont
               "/static/monogram.ttf" p
    sheet <- loadImage
               "/static/Cat_Grey.png" p
    put $ Just (font, sheet)

  (drawAddHandler, fireDraw) <-
    liftIO newAddHandler

  draw p $ do
    assets <- get @(Maybe Assets)
    mapM_ (liftIO . fireDraw) assets
```

2026-06-01 Effectful

An example that shows how to use the Effectful library: https://github.com/bcardiff/lambda-library/blob/main/app-effectful/Main.hs

2026-05-31 Weather System

I've always felt like the Zelda world is somehow more alive than the GTA world even though there are a lot more characters in the GTA world.

Perhaps it's the weather system. In Zelda the weather itself affects Link's health. It also has lunar cycles.

2026-05-31 RSS

There is now an RSS feed at ./rss.xml!

2026-05-28 Meow

Haskell kitty says meow.

Figure 2. Meow!

Cat asset

> https://bowpixel.itch.io/meow-cat-85-animation

Font asset

> https://datagoblin.itch.io/monogram