Abstractions for the Functional Roboticist

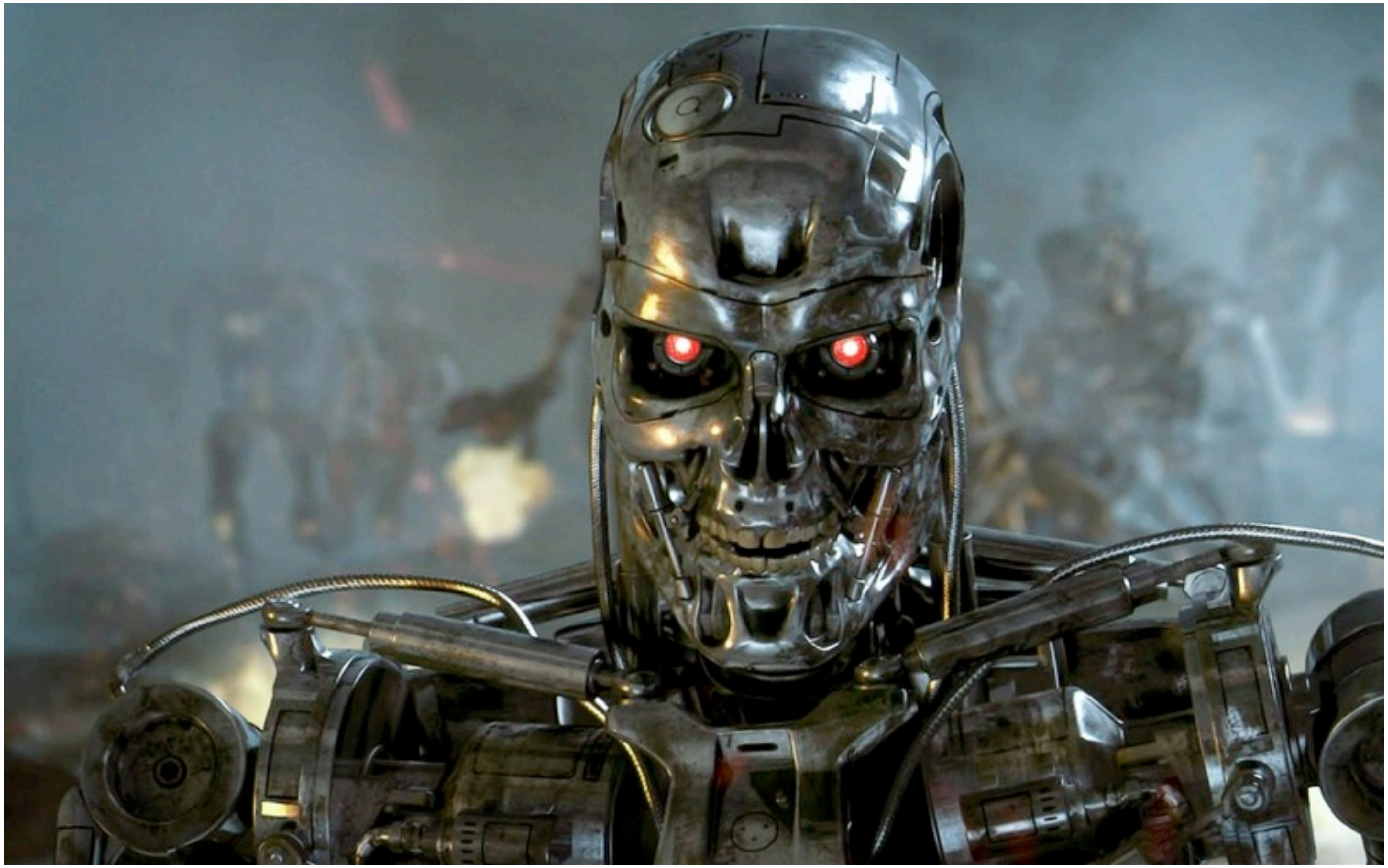
Anthony Cowley

"I'm using technology!"

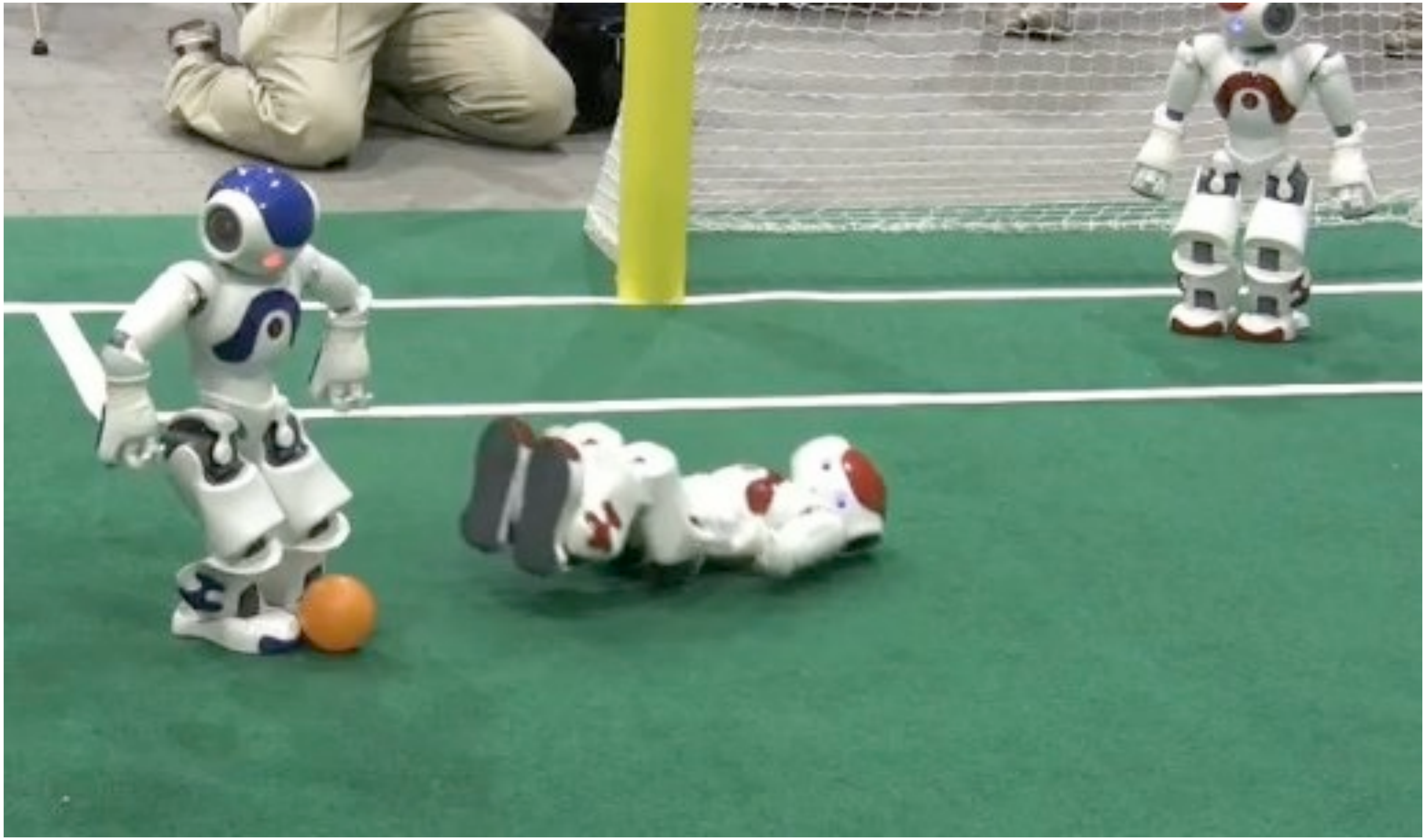
--Iggy Pop

Robotics Is...

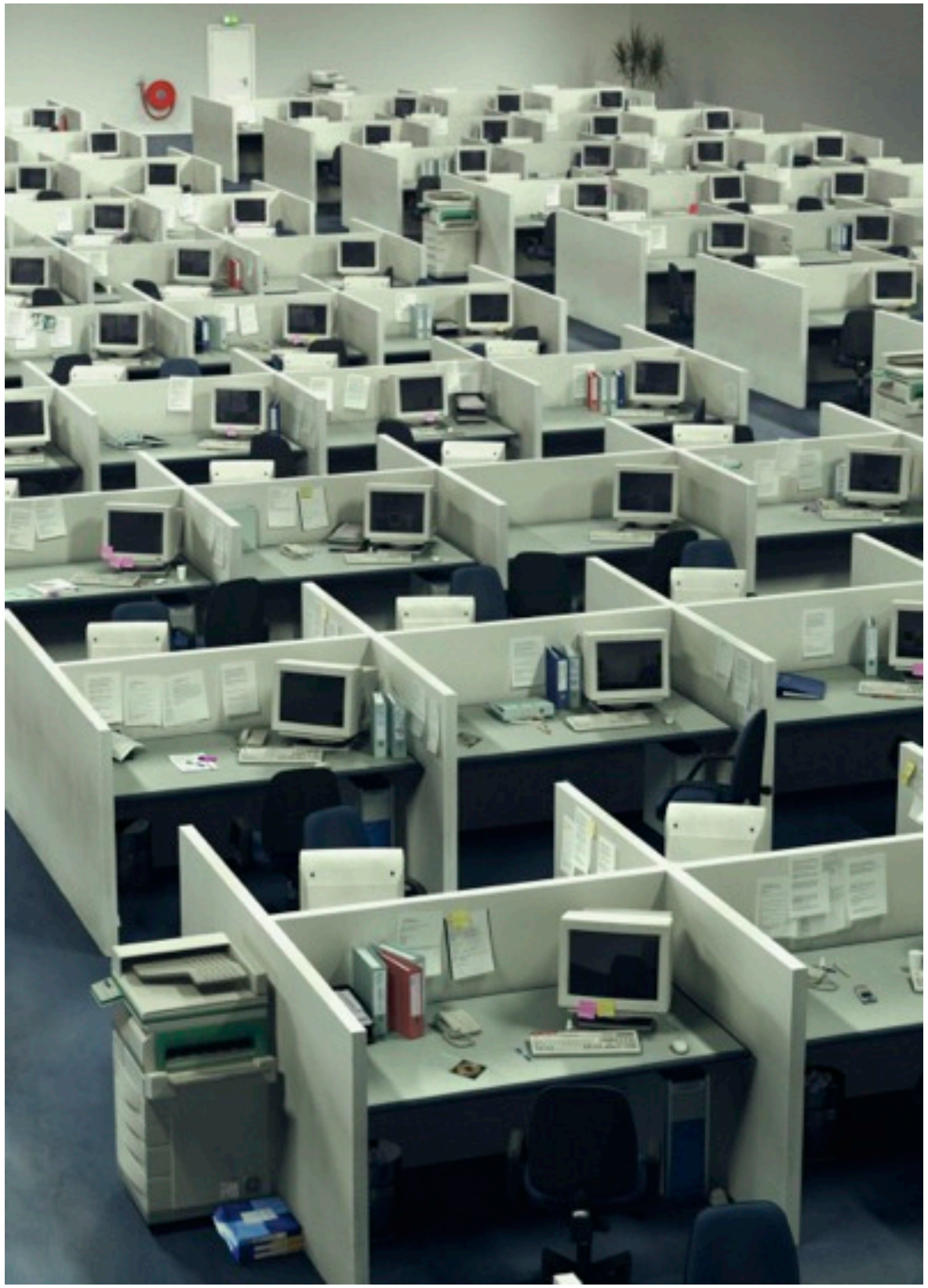
Robotics Is...



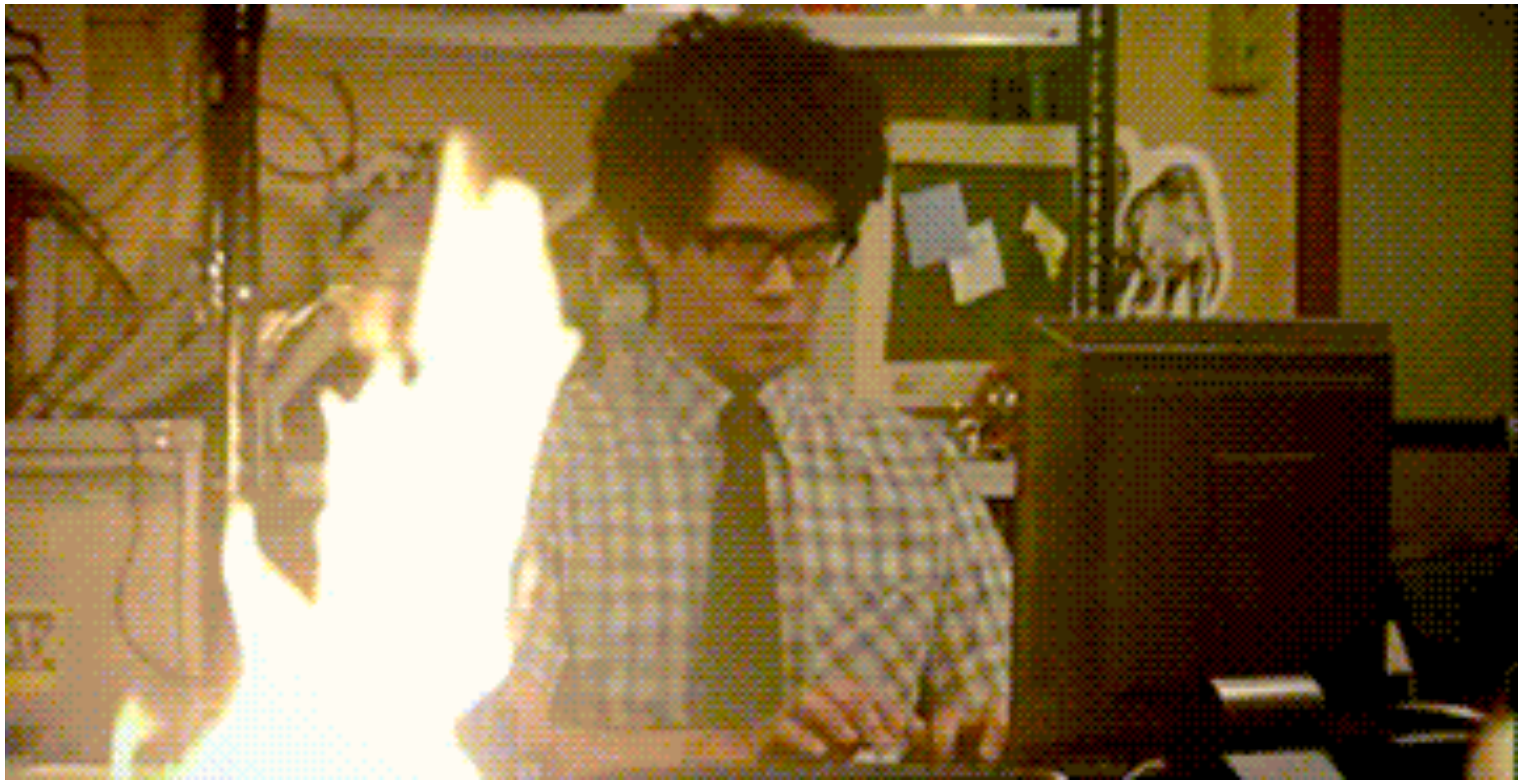
Robotics Is...



Robotics Is...



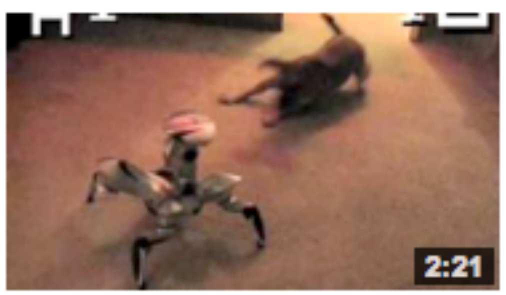
Robotics Is...



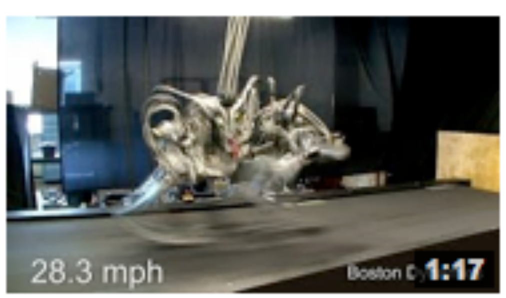
Taking Research Public

A Swarm of Nano Quadrotors
by **TheDmel** • 1 year ago • 7,356,064 views
Experiments performed with a team of nano quadrotors at the GRASP Lab, University of Pennsylvania. Vehicles developed by ...
HD

Sand Flea Jumping Robot
by **BostonDynamics** • 1 year ago • 6,126,974 views
Sand Flea is an 11-lb **robot** with one trick up its sleeve: Normally it drives like an RC car, but when it needs to it can jump 30 feet ...
HD



Puppy Vs. Robot! Epic Battle For Territorial Domination!
by **demonbabydotcom** • 5 years ago • 6,013,631 views
This is the **robot**: ...



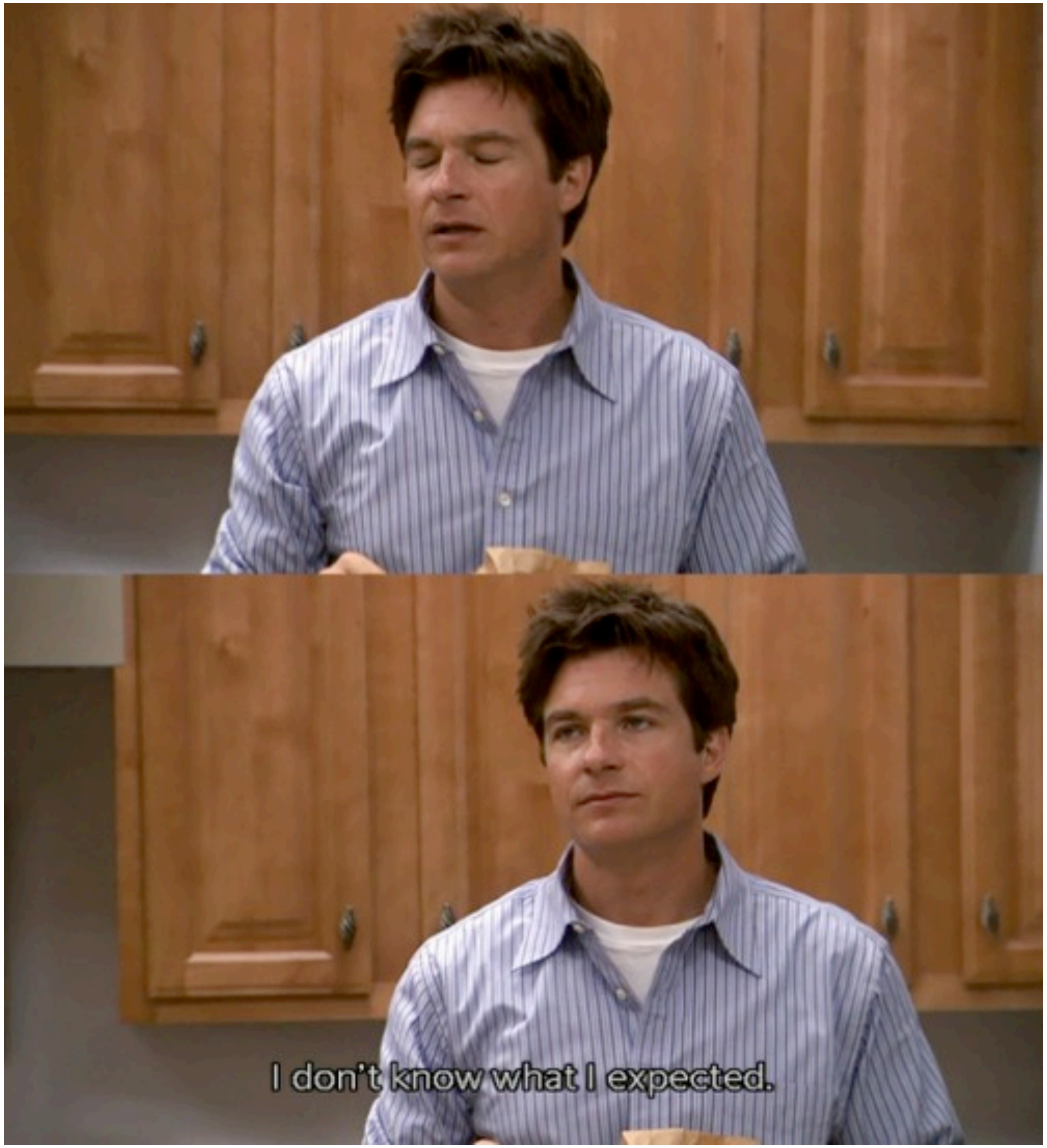
Cheetah Robot runs 28.3 mph; a bit faster than Usain Bolt
by **BostonDynamics** • 11 months ago • 5,765,497 views
Cheetah **Robot** is a fast-running quadruped developed by Boston Dynamics with funding from DARPA. It just blazed past its ...
HD

Big-box Abstractions

At Least it was Cheap

Let's Open the Box!

• • • •


I don't know what I expected.

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

--Edsgar Dijkstra

Hackage

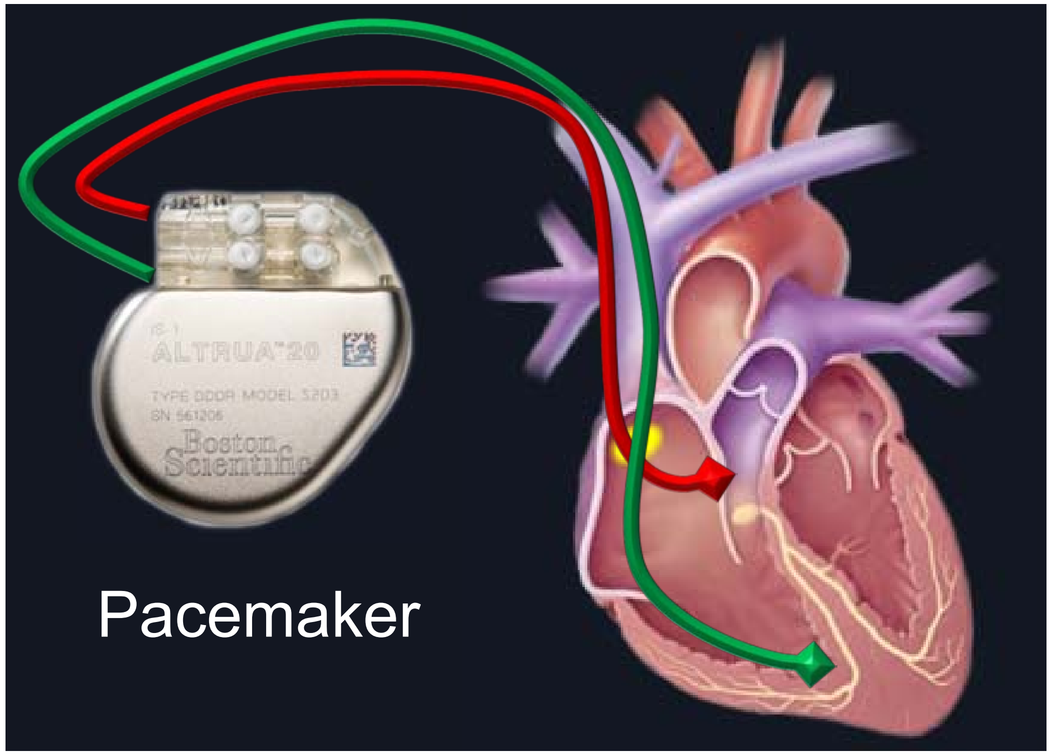
Nice Fit

Well, I wouldn't make that mistake

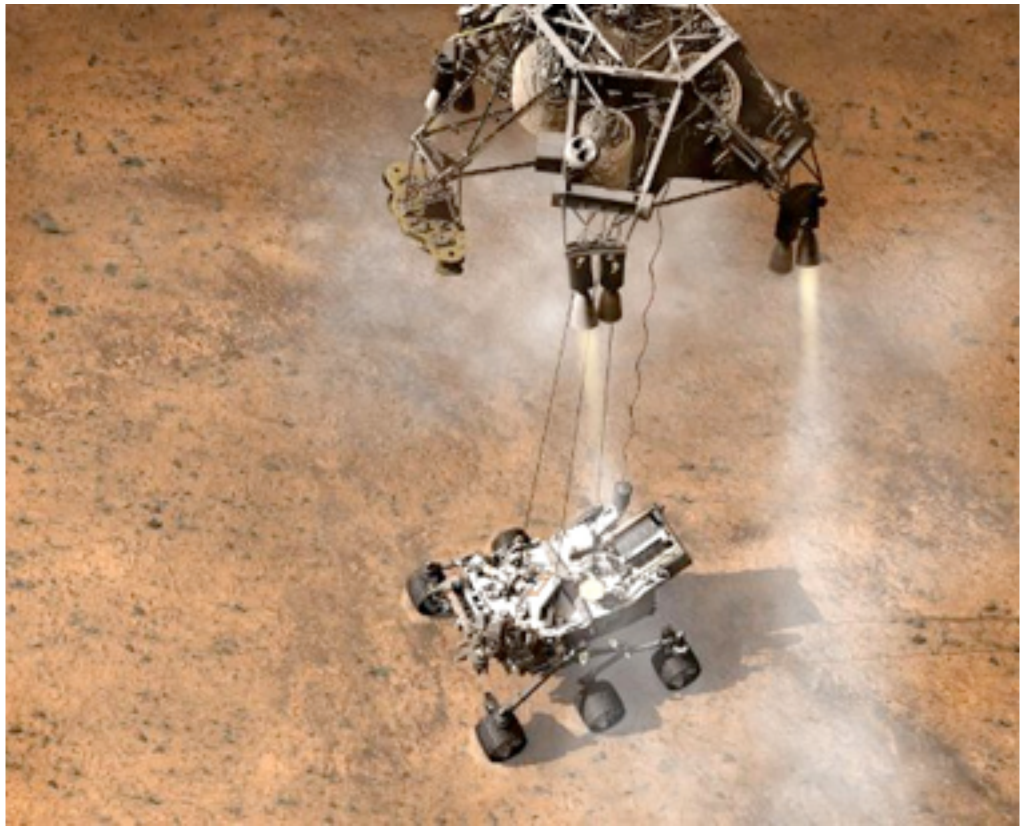
```
8     -     if missleFireTest():
   8  +     if not missleFireTest():
9  9            fire_missles()
```

Commit	Message
2784d9c	Bug fix.
d8e6ae6	Missle firing code

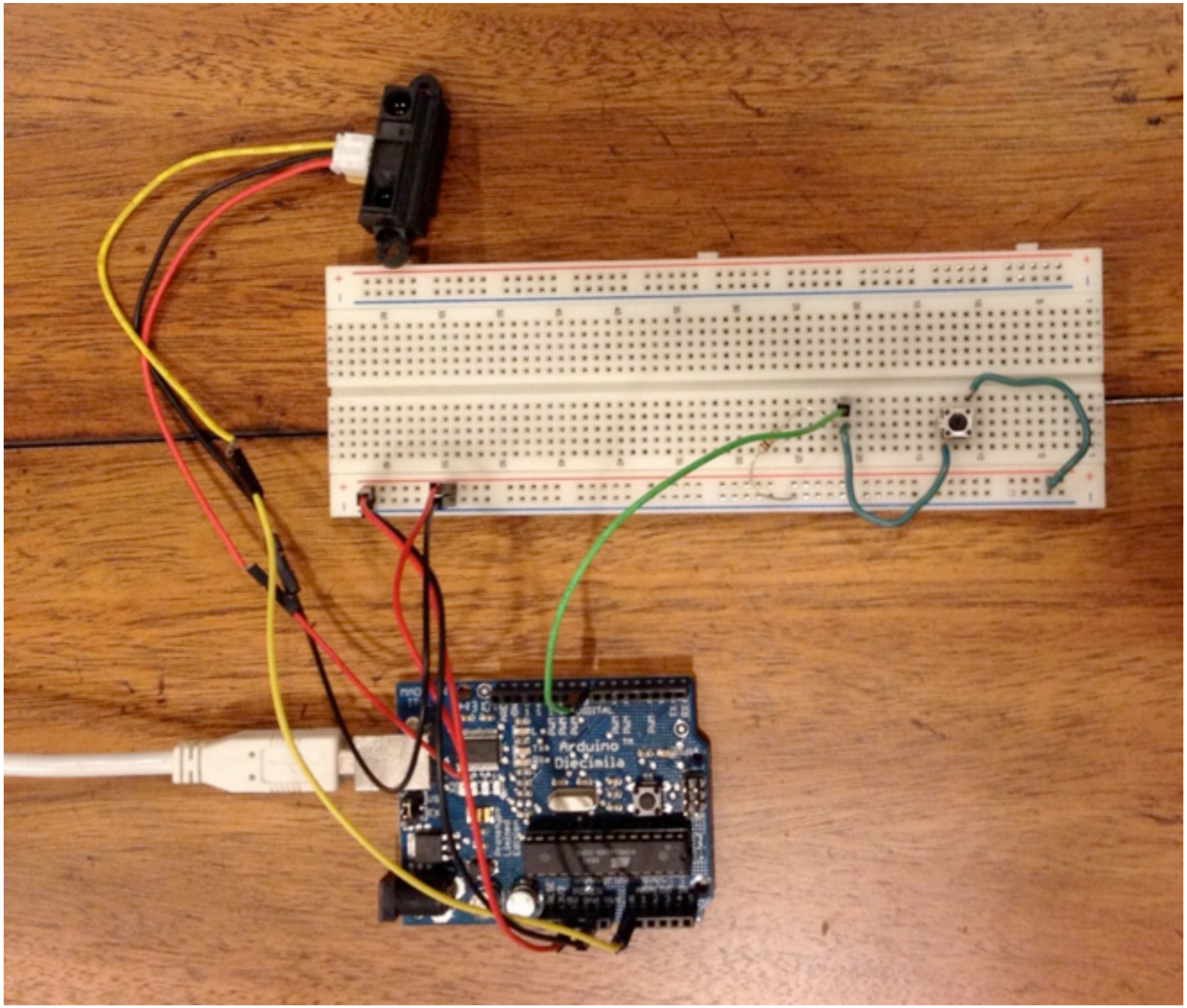
Correctness Counts



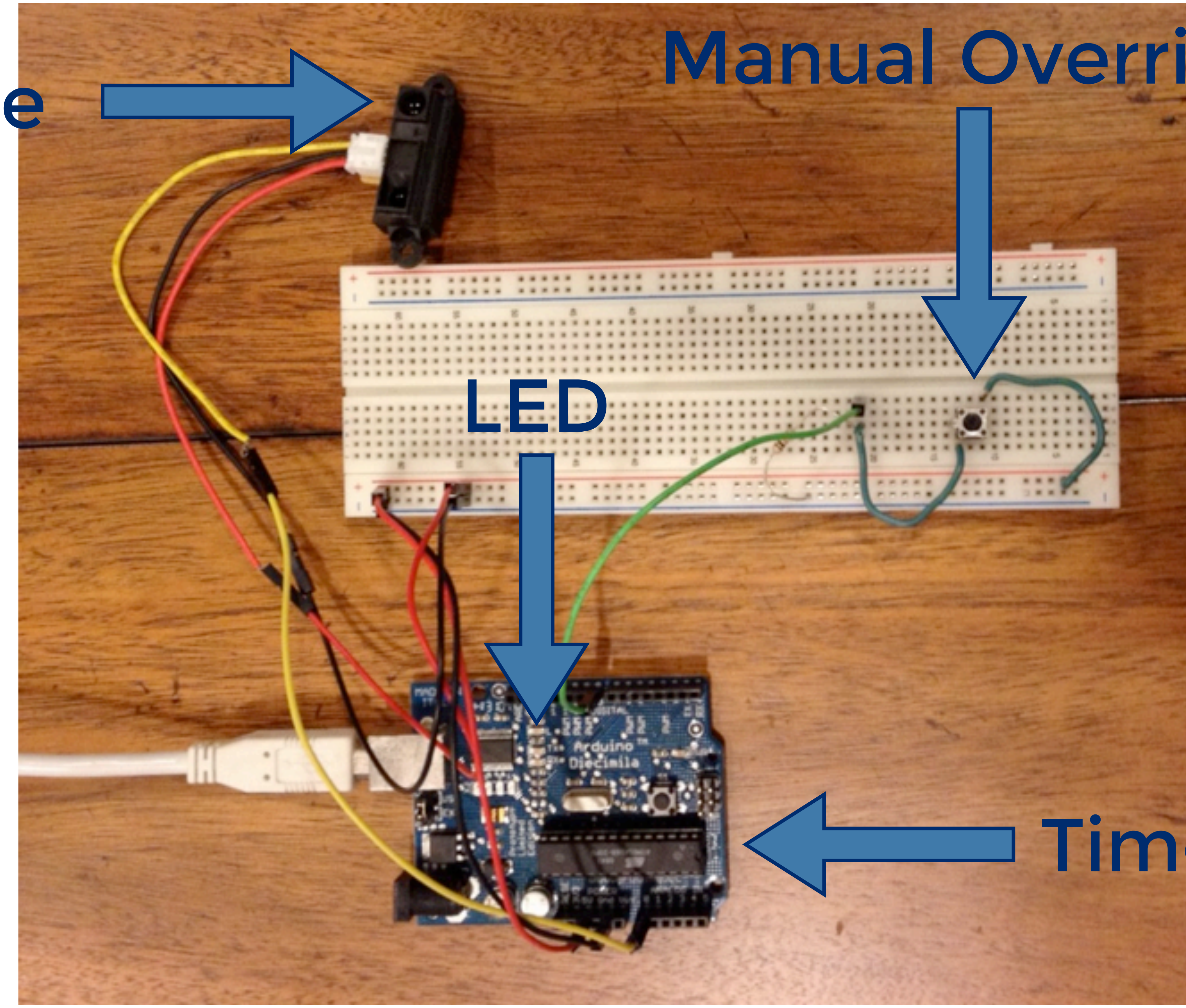
Credit: Zhihao Jiang, UPenn



Haskell for μC



Haskell for μC



Haskell for µC

- Spec
  - Send output if range > 200
  - Send output if manual override
  - Send one output for every 10 cycles no matter what

Haskell for μC

```haskell
module OutputGuard where
import Data.SBV

-- | Our range detector /must/ output if the range is closer than 200.
safetyDistance :: SWord16
safetyDistance = 200

-- | This is the specified limit. We want to stay below this!
maxTimeSince :: SWord16
maxTimeSince = 10

-- | Compute the time since the last output should have been sent. An
-- output of zero from this function means an output ought to be sent
-- right /now/. The goal is say that an output should be sent if the
-- range is greater than 200, the manual override is set, or we are
-- nearing the 'maxTimeSince' limit from the specification.
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 1)
      0
      (timeSince+1)
```

Check Your Logic at the Door

```haskell
pf1,pf2,pf3,spec :: Predicate
pf1 = forAll ["r", "m", "t"] $ \r m t -> f r m t .< maxTimeSince

pf2 = forAll ["r", "m", "t"] $ \r m t ->
        r .> safetyDistance ==> f r m t .== 0

pf3 = forAll ["r", "m", "t"] $ \r m t -> m ==> f r m t .== 0

spec = forAll ["r", "m", "t"] $ \r m t ->
       let minRate = f r m t .< maxTimeSince
           minRange = r .> safetyDistance ==> f r m t .== 0
           manualOverride = m ==> f r m t .== 0
       in minRate &&& minRange &&& manualOverride
```

Check Your Logic at the Door

```haskell
pf1,pf2,pf3,spec :: Predicate
pf1 = forAll ["r", "m", "t"] $ \r m t -> f r m t .< maxTimeSince

pf2 = forAll ["r", "m", "t"] $ \r m t ->
        r .> safetyDistance ==> f r m t .== 0

pf3 = forAll ["r", "m", "t"] $ \r m t -> m ==> f r m t .== 0

spec = forAll ["r", "m", "t"] $ \r m t ->
       let minRate = f r m t .< maxTimeSince
           minRange = r .> safetyDistance ==> f r m t .== 0
           manualOverride = m ==> f r m t .== 0
       in minRate &&& minRange &&& manualOverride
```

```
λ> prove pf1
Falsifiable. Counter-example:
  r = 0 :: SWord16
  m = False
  t = 9 :: SWord16
```

Check Your Logic at the Door

```haskell
-- | Compute the time since the last output should have been sent. An
-- output of zero from this function means an output ought to be sent
-- right /now/. The goal is say that an output should be sent if the
-- range is greater than 200, the manual override is set, or we are
-- nearing the 'maxTimeSince' limit from the specification.
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 1)
      0
      (timeSince+1)
```

Check Your Logic at the Door

```haskell
-- | Compute the time since the last output should have been sent. An
-- output of zero from this function means an output ought to be sent
-- right /now/. The goal is say that an output should be sent if the
-- range is greater than 200, the manual override is set, or we are
-- nearing the 'maxTimeSince' limit from the specification.
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 1)
      0
      (timeSince+1)
```

```haskell
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 2)
      0
      (timeSince+1)
```

Check Your Logic at the Door

```haskell
-- | Compute the time since the last output should have been sent. An
-- output of zero from this function means an output ought to be sent
-- right /now/. The goal is say that an output should be sent if the
-- range is greater than 200, the manual override is set, or we are
-- nearing the 'maxTimeSince' limit from the specification.
f :: SWord16 -[illegible]l -> SWord16 -> SWord16
f range manual timeS[illegible]
  ite (range .> 200 ||| man[illegible]imeSince .> maxTimeSince - 1)
      0
      (timeSince[illegible]
```

```haskell
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 2)
      0
      (timeSince+1)
```

```
λ> prove spec
Q.E.D.
```

Quick, Put it to Work!

```haskell
codeGen :: IO ()
codeGen = compileToC (Just "copilot-sbv-codegen") "outputGuard" $
          do cgGenerateDriver False
             cgGenerateMakefile False
             r <- cgInput "r"
             m <- cgInput "m"
             t <- cgInput "t"
             cgReturn $ f r m t
```

Quick, Put it to Work!

```haskell
codeGen :: IO ()
codeGen = compileToC (Just "copilot-sbv-codegen") "outputGuard" $
            do cgGenerateDriver False
               cgGenerateMakefile False
               r <- cgInput "r"
               m <- cgInput "m"
               t <- cgInput "t"
               cgReturn $ f r m t
```

```c
SWord16 outputGuard(const SWord16 r, const SBool m,
                    const SWord16 t)
{
  const SWord16 s0 = r;
  const SBool   s1 = m;
  const SWord16 s2 = t;
  const SBool   s4 = s0 > 0x00c8U;
  const SBool   s6 = s2 > 0x0008U;
  const SBool   s7 = s1 | s6;
  const SBool   s8 = s4 | s7;
  const SWord16 s11 = s2 + 0x0001U;
  const SWord16 s12 = s8 ? 0x0000U : s11;

  return s12;
}
```

That was... easy

```haskell
module OutputGuard where
import Data.SBV

-- | Our range detector /must/ output if the range is closer than 200.
safetyDistance :: SWord16
safetyDistance = 200

-- | This is the specified limit. We want to stay below this!
maxTimeSince :: SWord16
maxTimeSince = 10

-- | Compute the time since the last output should have been sent. An
-- output of zero from this function means an output ought to be sent
-- right /now/. The goal is say that an output should be sent if the
-- range is greater than 200, the manual override is set, or we are
-- nearing the 'maxTimeSince' limit from the specification.
f :: SWord16 -> SBool -> SWord16 -> SWord16
f range manual timeSince =
  ite (range .> 200 ||| manual ||| timeSince .> maxTimeSince - 2)
      0
      (timeSince+1)

pf1,pf2,pf3,spec :: Predicate
pf1 = forAll ["r", "m", "t"] $ \r m t -> f r m t .< maxTimeSince

pf2 = forAll ["r", "m", "t"] $ \r m t ->
        r .> safetyDistance ==> f r m t .== 0

pf3 = forAll ["r", "m", "t"] $ \r m t -> m ==> f r m t .== 0

spec = forAll ["r", "m", "t"] $ \r m t ->
       let minRate = f r m t .< maxTimeSince
           minRange = r .> safetyDistance ==> f r m t .== 0
           manualOverride = m ==> f r m t .== 0
       in minRate &&& minRange &&& manualOverride

codeGen :: IO ()
codeGen = compileToC (Just "copilot-sbv-codegen") "outputGuard" $
          do cgGenerateDriver False
             cgGenerateMakefile False
             r <- cgInput "r"
             m <- cgInput "m"
             t <- cgInput "t"
             cgReturn $ f r m t
```

Crash Course in Copilot

It's just a bunch of streams

```haskell
import Language.Copilot
import qualified Copilot.Compile.SBV as S
import qualified Prelude as P
import qualified OutputGuard as OG

ledPin, rangePin, buttonPin :: Int16
ledPin    = 13
rangePin  = 0
buttonPin = 8

outputGuard :: Stream Word16 -> Stream Bool -> Stream Word16 -> Stream Word16
outputGuard r m p = externFun "outputGuard" [arg r, arg m, arg p] Nothing

analogRead :: Int16 -> Stream Word16
analogRead p = [0] ++ externFun "analogRead" [arg $ constI16 p] Nothing

digitalRead :: Int16 -> Stream Bool
digitalRead p = [False] ++ externFun "digitalRead" [arg $ constI16 p] Nothing
```

Crash Course in Copilot

It's just a bunch of streams

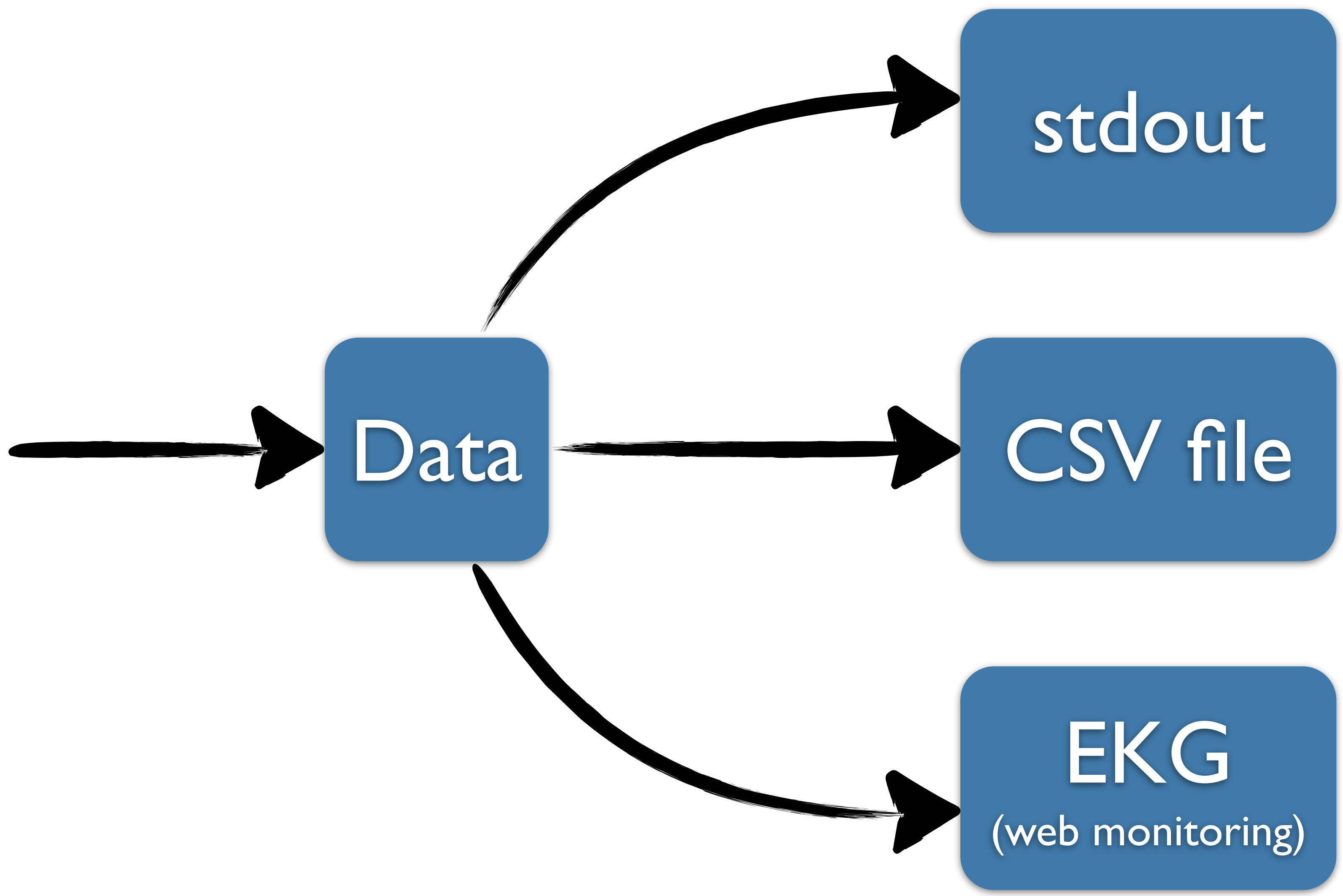
```
-- We have to be careful about only depending on yesterday's values
-- for today's logic!
rangeReporter :: Spec
rangeReporter = do trigger "serialPrint" go [arg range]
                   trigger "digitalWrite" true [arg $ constI16 ledPin, arg go]
  where range = analogRead rangePin
        manual = digitalRead buttonPin
        prev = [2] ++ outputGuard range manual prev
        go = [False] ++ prev == 0

-- Compile 'rangeReporter' to C using the SBV backend. This creates a
-- @step@ function that we want to call in our main loop.
toSBV :: IO ()
toSBV = reify rangeReporter >>= S.compile S.defaultParams
```

Crash Course in Copilot

Arduino Specifics

```haskell
setup :: String
setup = P.unlines [ "#include <Arduino.h>"
                  , "#include <stdint.h>"
                  , "void setup() {"
                  , "  pinMode(" P.++ show ledPin P.++ ", OUTPUT);"
                  , "  pinMode(" P.++ show rangePin P.++ ", INPUT);"
                  , "  pinMode(" P.++ show buttonPin P.++", INPUT);"
                  , "  Serial.begin(9600);"
                  , "}" ]

loop :: String
loop = P.unlines [ "extern \"C\" void step(void);"
                 , "extern \"C\" void serialPrint(uint16_t x) {"
                 , "  Serial.println(x);"
                 , "}"
                 , "void loop() {"
                 , "  step();"
                 , "  delay(100);"
                 , "}" ]

fixIno :: IO ()
fixIno = writeFile "arduino/src/sketch.cpp" (setup P.++ loop)

main :: IO ()
main = OG.codeGen >> toSBV >> fixIno
```

Compositional Logging



Crash Course in Machines

It's just a bunch of streams

```
newtype MachineT m k o                                        Source
```

A `MachineT` reads from a number of inputs and may yield results before stopping with monadic side-effects.

Constructors

```
MachineT
    runMachineT :: m (Step k o (MachineT m k o))
```

```
data Step k o r
```

This is the base functor for a `Machine` or `MachineT`.

Note: A `Machine` is usually constructed from `Plan`, so it does not need to be CPS'd.

Constructors

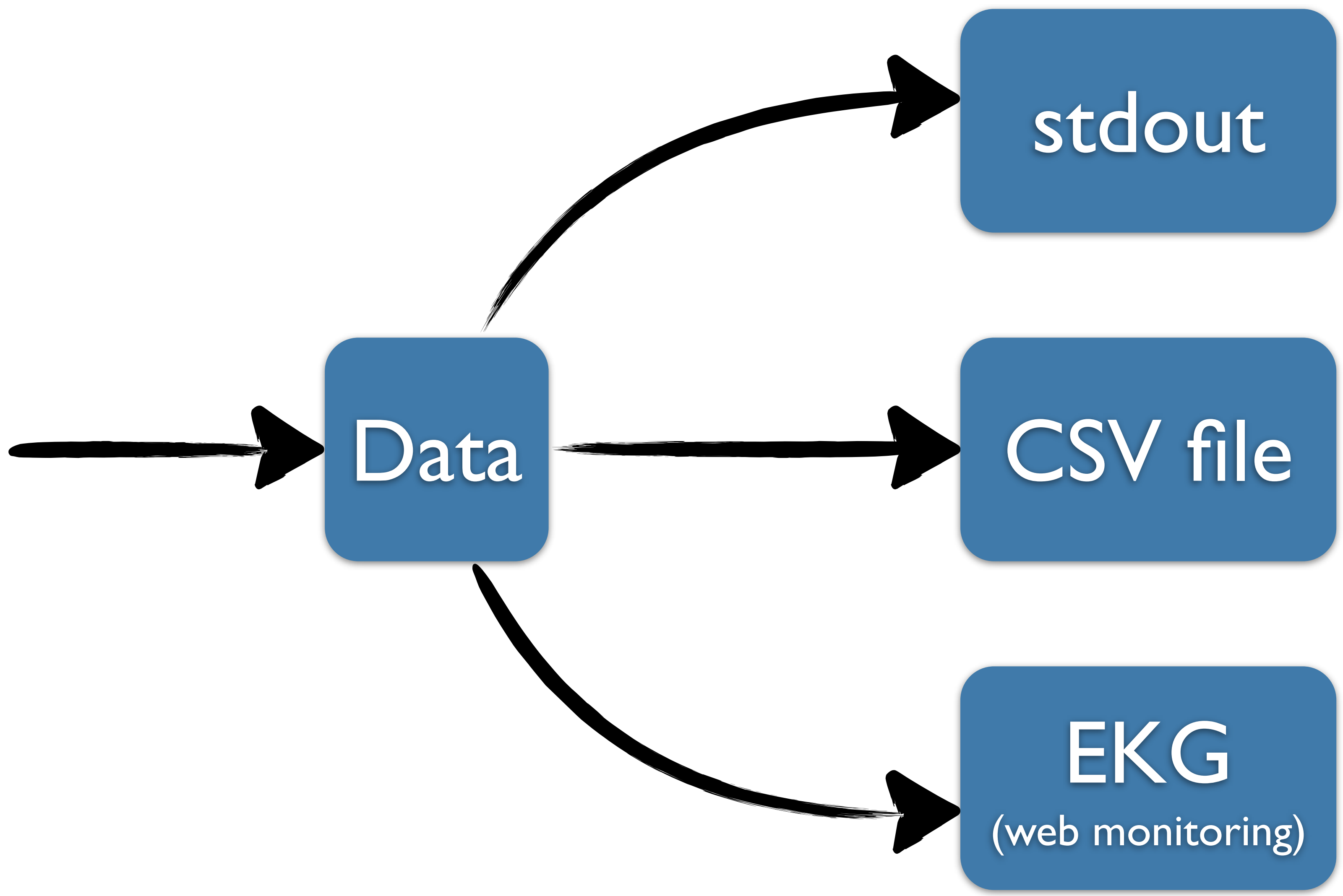
```
Stop
Yield o r
forall t . Await (t -> r) (k t) r
```

Crash Course in Serial Ports

It's just a ~~bunch of~~ stream~~s~~

```haskell
{-# LANGUAGE OverloadedStrings, RankNTypes #-}
module Monitor where
import Control.Applicative
import Control.Monad.IO.Class
import qualified Data.ByteString.Char8 as B
import Data.Machine
import System.Hardware.Serialport
import Text.Read

serialMachine :: MonadIO m => SerialPort -> SourceT m B.ByteString
serialMachine p = repeatedly $ liftIO (recv p 8) >>= yield

-- | Break into lines separated by "\r\n" characters.
rnLines :: B.ByteString -> [B.ByteString]
rnLines bs
  | B.null t = [h]
  | otherwise = h : rnLines (B.drop 2 t)
  where (h,t) = B.breakSubstring "\r\n" bs

readMachine :: Read a => Process String a
readMachine = repeatedly $ await >>= maybe (return ()) yield . readMaybe

asciiLines :: Process B.ByteString String
asciiLines = construct $ go B.empty
  where go xs = do i <- B.filter (/= '\0') <$> await
                   let lns = rnLines $ B.append xs i
                   mapM_ (yield . B.unpack) $ init lns
                   go $ last lns

monitor :: MonadIO m => SerialPort -> SourceT m String
monitor p = serialMachine p ~> asciiLines
```

Compositional Logging



Compositional Logging

```haskell
{-# LANGUAGE GADTs, OverloadedStrings #-}
module Logger (logger) where
import Control.Applicative
import Control.Monad
import Control.Monad.IO.Class
import Data.List
import Data.Machine
import Data.Maybe
import Data.Void
import System.Remote.Monitoring
import qualified System.Remote.Gauge as G

-- | Build a composite logging process by feeding individual loggers
-- in lockstep.
mkLogger :: (Functor m, MonadIO m) => [ProcessT m Int Void] -> ProcessT m Int r
mkLogger loggers = encased $ Await (MachineT . aux) Refl (mkLogger loggers)
  where aux x = mapM (runMachineT >=> feed) loggers
                >>= runMachineT . mkLogger . catMaybes
          where feed (Await f Refl _) = Just . encased <$> runMachineT (f x)
                feed (Yield _ _)      = error "Impossible to yield a Void!"
                feed Stop             = return Nothing

-- | Print ranges greater than 100 to stdout.
logStdOut :: MonadIO m => ProcessT m Int Void
logStdOut = repeatedly $
            do r <- await
               when (r > 100) (liftIO . putStrLn $ "Range = "++show r)

-- | Log all values to a file on disk in batches of 50.
logCSV :: MonadIO m => FilePath -> ProcessT m Int Void
logCSV f = buffered 50
         ~> construct (liftIO (writeFile f "ranges\n") >> forever go)
  where go = await >>= liftIO . flushLines
        flushLines = appendFile f . (++"\n") . intercalate "\n" . map show

-- | Start up an EKG monitor, and update a 'Gauge' with the current
-- range.
logEKG :: MonadIO m => ProcessT m Int Void
logEKG = construct $ liftIO setup >>= forever . go
  where setup = forkServer "localhost" 8000 >>= getGauge "Range"
        go g = await >>= liftIO . G.set g

-- | The composite logger for range information.
logger :: (Functor m, MonadIO m) => ProcessT m Int r
logger = mkLogger [logStdOut, logCSV "ranges.csv", logEKG]
```

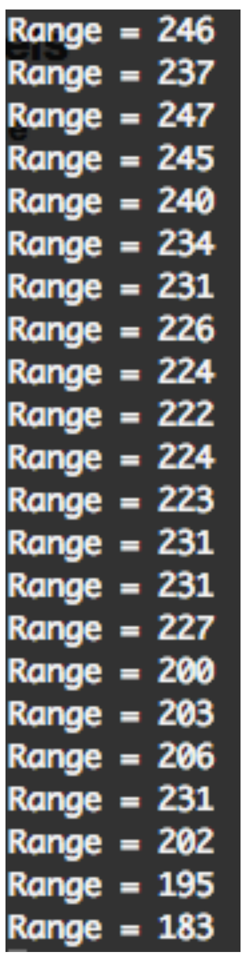
Compositional Logging

```haskell
-- | Print ranges greater than 100 to stdout.
logStdOut :: MonadIO m => ProcessT m Int Void
logStdOut = repeatedly $
              do r <- await
                 when (r > 100) (liftIO . putStrLn $ "Range = "++show r)

-- | Log all values to a file on disk in batches of 50.
logCSV :: MonadIO m => FilePath -> ProcessT m Int Void
logCSV f = buffered 50
          ~> construct (liftIO (writeFile f "ranges\n") >> forever go)
  where go = await >>= liftIO . flushLines
        flushLines = appendFile f . (++"\n") . intercalate "\n" . map show

-- | Start up an EKG monitor, and update a 'Gauge' with the current
-- range.
logEKG :: MonadIO m => ProcessT m Int Void
logEKG = construct $ liftIO setup >>= forever . go
  where setup = forkServer "localhost" 8000 >>= getGauge "Range"
        go g = await >>= liftIO . G.set g

-- | The composite logger for range information.
logger :: (Functor m, MonadIO m) => ProcessT m Int r
logger = mkLogger [logStdOut, logCSV "ranges.csv", logEKG]
```

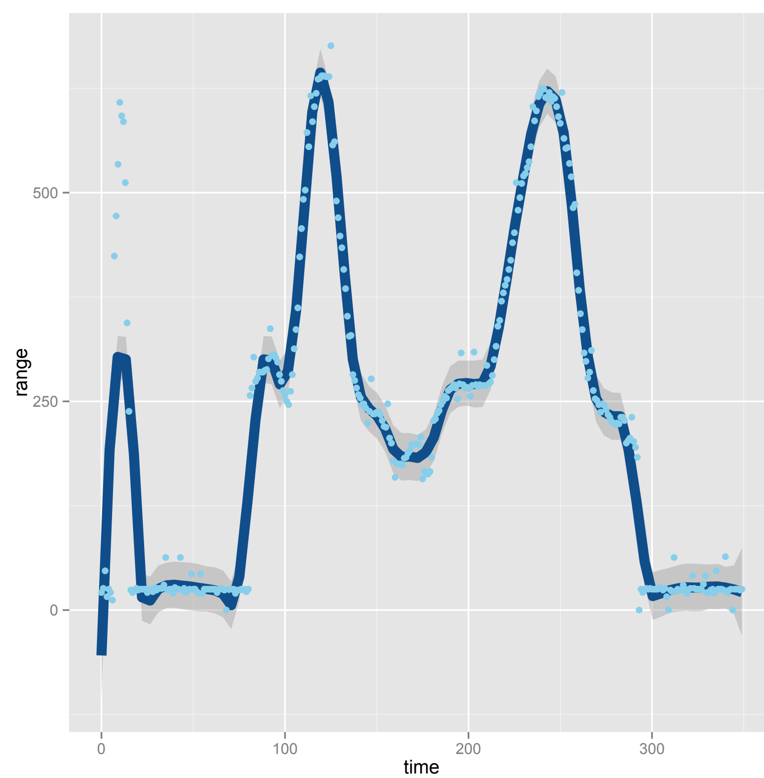
Final Tally

that’s a lot of functionality

151 LOC



stdout



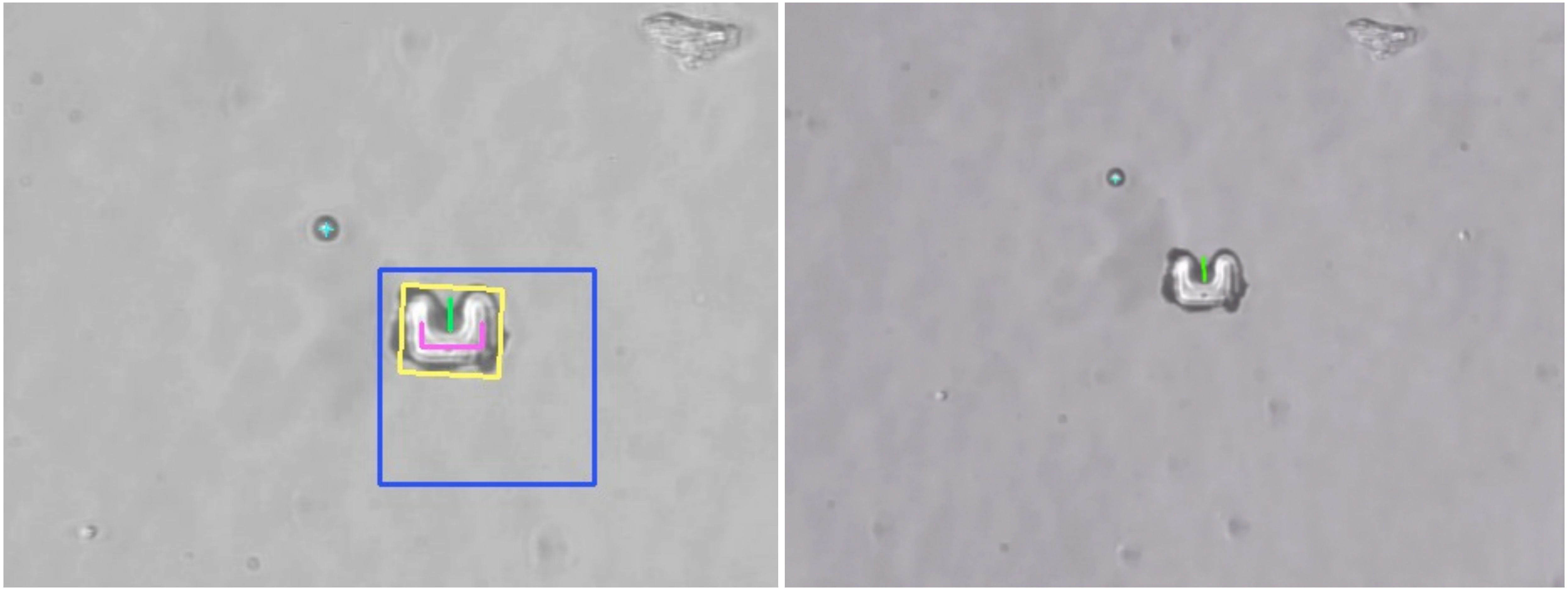
CSV → R + ggplot2

EKG

From microcontrollers
to
micro robots

Microtransporter

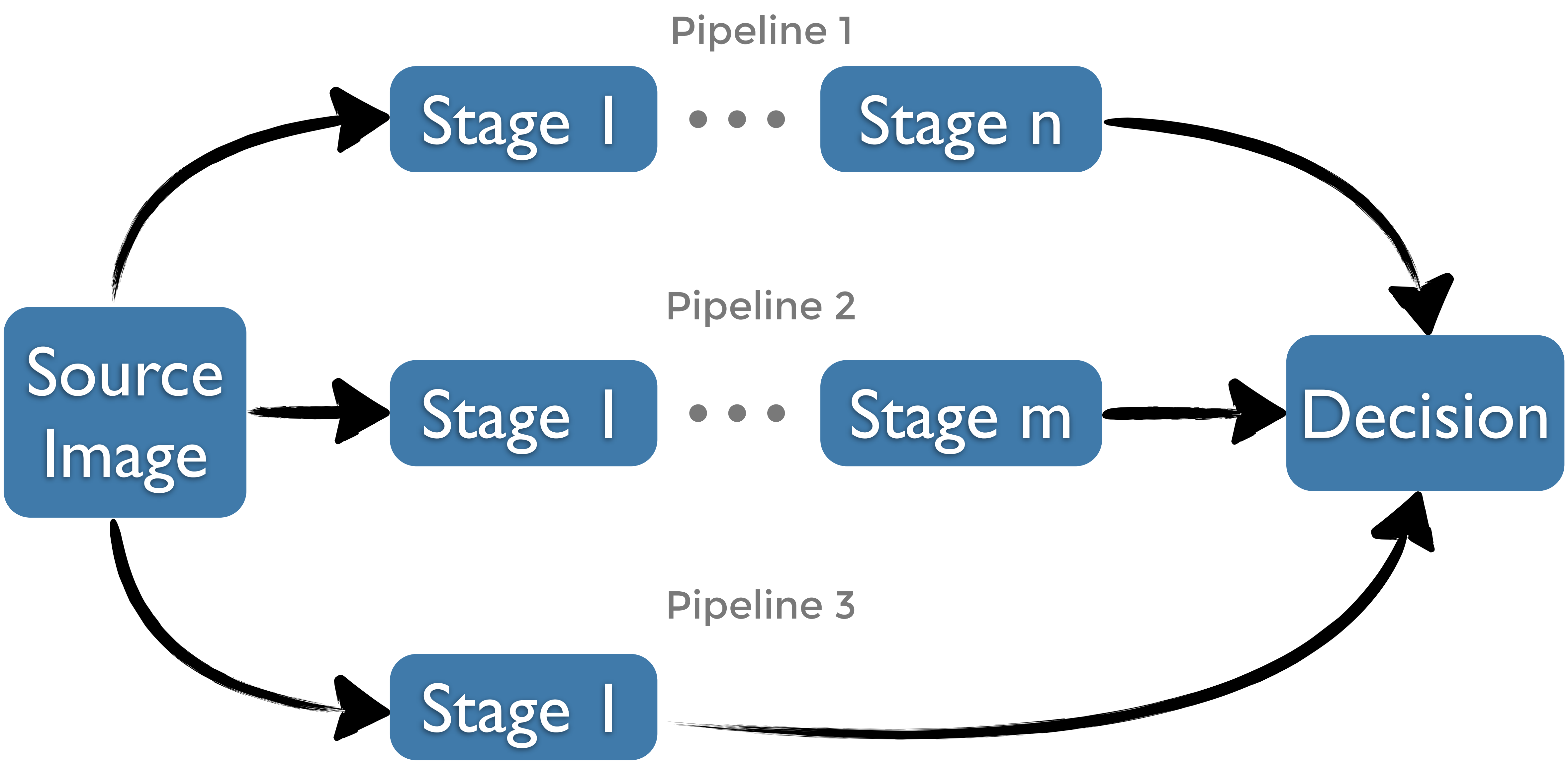
Robot is about 0.04mm



Sakar, et al,
“Wireless manipulation of single cells using magnetic microtransporters”
ICRA 2011

Computer Vision

It's just a bunch of pipelines



Persistent Data

```haskell
-- When working with the FFI, we will often have a 'Ptr' to our data.
type MyData = IORef Int

myData :: Int -> MyData
myData = unsafePerformIO . newIORef
{-# NOINLINE myData #-}
```

Persistent Data

```
-- When working with the FFI, we will often have a 'Ptr' to our data.
type MyData = IORef Int

myData :: Int -> MyData
myData = unsafePerformIO . newIORef
{-# NOINLINE myData #-}

-- The operation we're calling through the FFI mutates its argument
-- in-place.
foreignOp :: MyData -> IO ()
foreignOp = flip modifyIORef' succ
```

Persistent Data

```haskell
-- When working with the FFI, we will often have a 'Ptr' to our data.
type MyData = IORef Int

myData :: Int -> MyData
myData = unsafePerformIO . newIORef
{-# NOINLINE myData #-}

-- The operation we're calling through the FFI mutates its argument
-- in-place.
foreignOp :: MyData -> IO ()
foreignOp = flip modifyIORef' succ

-- So when we want to provide a pure interface, we have to manually
-- create a copy of the input.
dupData :: MyData -> IO MyData
dupData = readIORef >=> newIORef

op :: (MyData -> IO ()) -> MyData -> MyData
op f x = unsafePerformIO $
         do x' <- threadDelay 1000000 >> dupData x
            readIORef x >>= putStrLn . ("Duplicated "++) . show
            f x'
            return x'
{-# NOINLINE op #-}

-- Now we can get a handle on a pure version of our operation that
-- prevents in-place mutation.
inc :: MyData -> MyData
inc = op foreignOp
{-# INLINE inc #-}
```

Paying for Persistence

```haskell
showData :: MyData -> String

main :: IO ()
main = do let x = myData 3
          putStrLn $ "x = " ++ showData x
          let y = inc . inc $ x
          putStrLn $ "y = " ++ showData y
```

```
$ ./Inplace
x = 3
Duplicated 3
Duplicated 4
y = 5
```

What happens in a composition, stays in a composition

```
showData :: MyData -> String

main :: IO ()
main = do let x = myData 3
          putStrLn $ "x = " ++ showData x
          let y = inc . inc $ x
          putStrLn $ "y = " ++ showData y
```

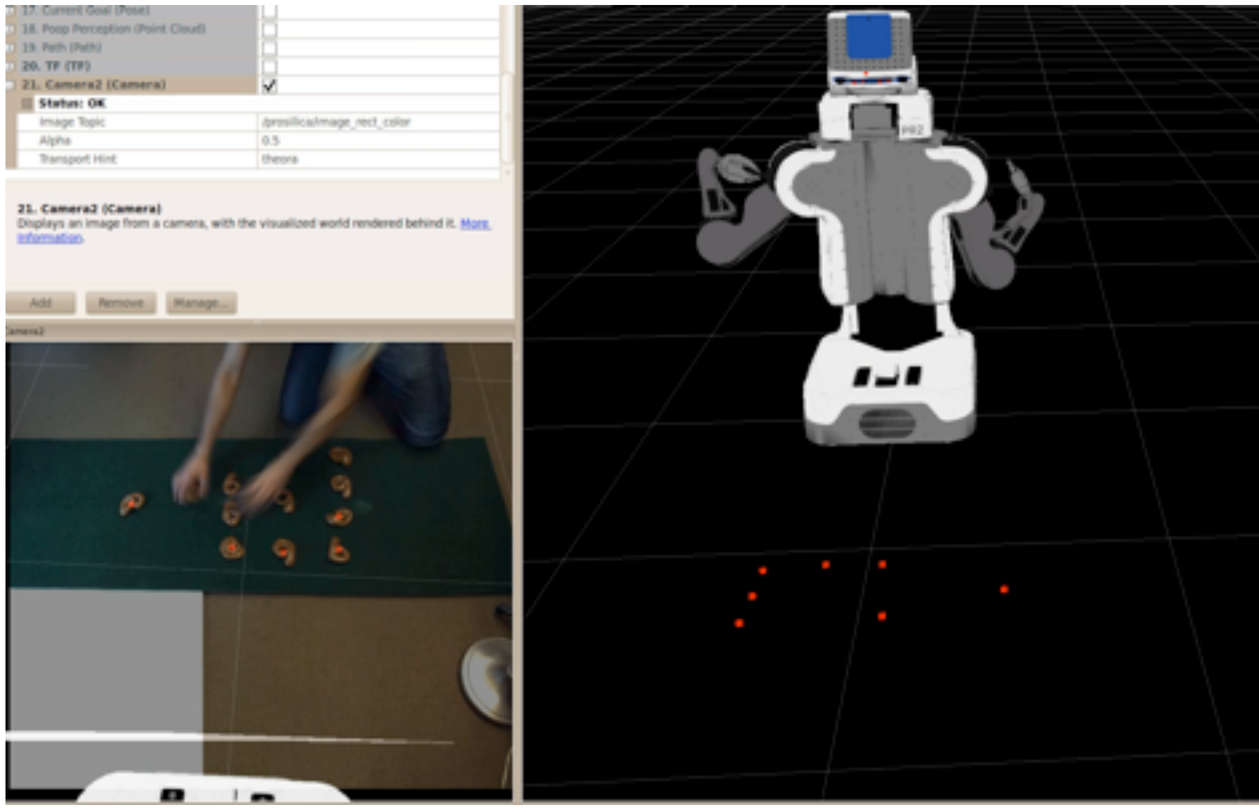
```
$ ./Inplace
x = 3
Duplicated 3
Duplicated 4
y = 5
```

```
{-# RULES "op/compose" forall f g. op f . op g = op (\x -> g x >> f x) #-}
```

```
25 RuleFired
  5 Class op >>
  3 ++
  3 Class op >>=
  3 unpack
  2 Class op show
  2 fold/build
  2 unpack-append
  1 Class op return
  1 Class op succ
  1 foldr/app
  1 op/compose
  1 unpack-list
```

```
$ ./Inplace
x = 3
Duplicated 3
y = 5
```

Happy Little Accidents



I was told there'd be GPUs

GLBoilerPlate

```haskell
{-# LANGUAGE DataKinds, TypeOperators #-}
import Data.Proxy
import Data.Vinyl
import Graphics.VinylGL
import Graphics.Rendering.OpenGL (GLfloat)
import Linear

type TexCoord = "texCoord" ::: V2 GLfloat

texCoord :: TexCoord
texCoord = Field

type Vert = PlainRec ["vertexPos" ::: V3 GLfloat, TexCoord]

main :: IO ()
main = print $ fieldToVAD texCoord (Proxy::Proxy Vert)
```

GLBoilerPlate

```haskell
{-# LANGUAGE DataKinds, TypeOperators #-}
import Data.Proxy
import Data.Vinyl
import Graphics.VinylGL
import Graphics.Rendering.OpenGL (GLfloat)
import Linear

type TexCoord = "texCoord" ::: V2 GLfloat

texCoord :: TexCoord
texCoord = Field

type Vert = PlainRec ["vertexPos" ::: V3 GLfloat, TexCoord]

main :: IO ()
main = print $ fieldToVAD texCoord (Proxy::Proxy Vert)
```

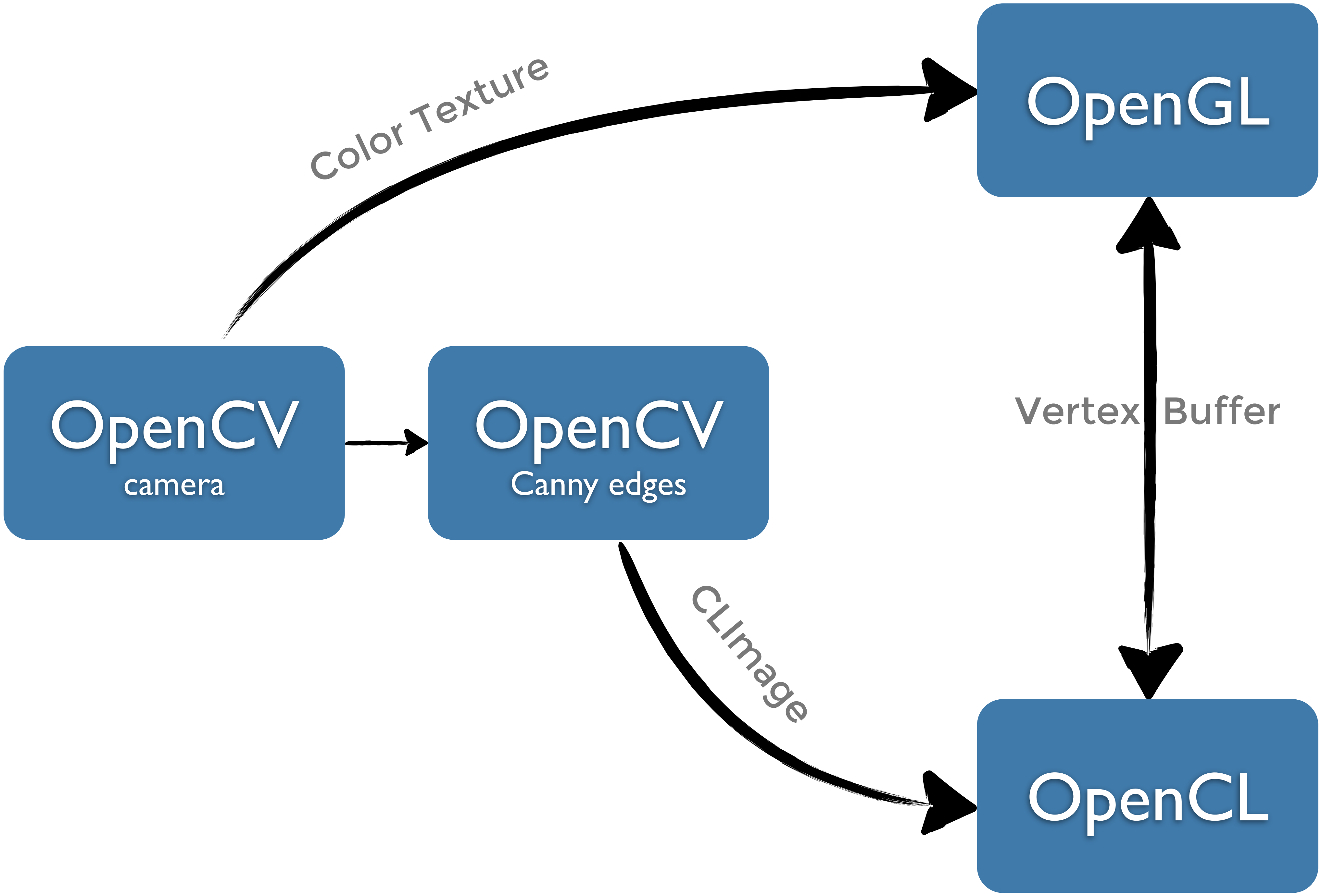
```
> VertexArrayDescriptor 2 Float 20 0x000000000000000c
```

For more, see: http://www.arcadianvisions.com/blog/?p=388

Crash course in OpenCL

```haskell
import Control.Monad
import Control.Parallel.CLUtil
import Control.Parallel.CLUtil.Monad
import qualified Data.Vector.Storable as V

mkVadd :: Int -> CL (Vector CFloat -> Vector CFloat -> CL (Vector CFloat))
mkVadd n = do k <- getKernel "VecAdd.cl" "vadd1D"
              [b1,b2] <- replicateM 2 $ allocBuffer [CL_MEM_READ_ONLY] n
              r <- allocBuffer [CL_MEM_WRITE_ONLY] n
              return $ \v1 v2 ->
                do writeBuffer b1 v1
                   writeBuffer b2 v2
                   () <- runKernelCL k b1 b2 r (Work1D n)
                   readBuffer r

main :: IO ()
main = do gpu <- ezInit CL_DEVICE_TYPE_GPU
          vadd <- runCL gpu $ mkVadd 6
          r <- runCL gpu $ vadd v1 v2
          showOutput r
  where v1 = V.fromList [1,2,3,4,5,6]
        v2 = V.fromList [7,8,9,10,11,12]
```

Open*

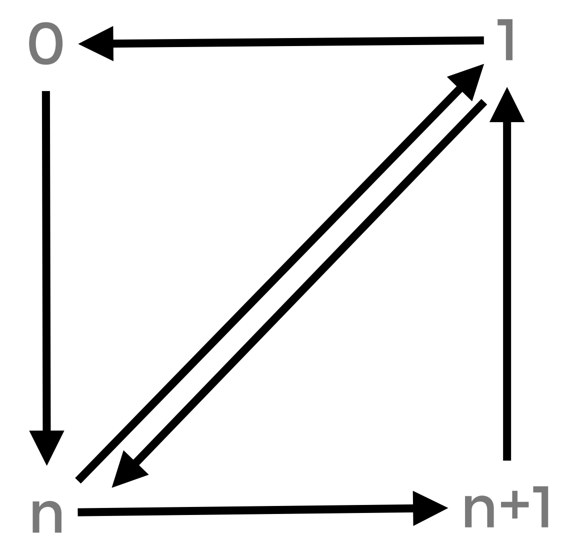

OpenGL
Color Texture
OpenCV
camera
OpenCV
Canny edges
Vertex Buffer
CLImage
OpenCL

You've Got a Stew Going

Preliminaries

```haskell
{-# LANGUAGE DataKinds, TypeOperators, TupleSections #-}
import Control.Applicative
import Control.Parallel.CLUtil
import Control.Parallel.CLUtil.Monad
import CLGLInterop
import qualified Data.Vector.Storable as V
import Data.Vinyl
import Foreign.Ptr (nullPtr)
import Graphics.GLUtil
import Graphics.GLUtil.Camera3D
import Graphics.Rendering.OpenGL
import Graphics.UI.GLFW (Key(KeyEsc))
import Linear
import OpenCV.HighCV
import OpenCV.PixelUtils (packPixels)
import Keyboard3D (moveCamera)
import Graphics.VinylGL
import Window (initGL, UI(..))
```

Geometry




```haskell
type AppInfo = PlainRec '[ "proj"  ::: M44 GLfloat ]
type Pos = "vertexPos" ::: V4 GLfloat

pos :: Pos
pos = Field

-- | Uniform grid of vertices with @n*2@ vertices on a side.
grid :: Integral a => a -> [V2 GLfloat]
grid n = map (fmap ((*s) . fromIntegral)) [V2 x y | y <- [-n..n], x <- [-n..n]]
  where s = 1 / fromIntegral n

-- | Indices of an @n@ x @n@ grid in row-major order.
inds :: Word32 -> [Word32]
inds n = take (numQuads*6) $ concat $ iterate (map (n+)) row
  where numQuads = fromIntegral $ (n - 1) * (n - 1)
        row = concatMap (\c -> map (c+) [0,n,1,1,n,n+1]) [0..n - 2]
```

OpenCL and OpenCV

```haskell
ripple :: BufferObject -> CL (Vector Word8 -> CL (), IO ())
ripple b = do verts <- bufferFromGL b :: CL (CLBuffer (V4 Float))
              let workSize = Work1D $ bufferLength verts
                  sz = [640,480] :: [Int]
              img <- allocImage [CL_MEM_READ_ONLY] sz :: CL (CLImage1 Float)
              img' <- allocImage [CL_MEM_READ_WRITE] sz :: CL (CLImage1 Float)
              let cleanup = () <$ (releaseObject verts >> releaseObject img)
              k <- getKernel "etc/ripples.cl" "ripple"
              blur <- mkBlur
              return . (,cleanup) $ \e ->
                do writeImage img (V.map ((/255) . fromIntegral) e)
                   blur img img'
                   withGLObjects [bufferObject verts] $
                     runKernelCL k img' verts workSize

mkBlur :: CL (CLImage n a -> CLImage n a -> CL ())
mkBlur = do k <- getKernel "etc/ripples.cl" "localMax"
            return $ \i b -> do () <- runKernelCL k i b stepX n w
                                runKernelCL k i b stepY n w
  where stepX = V2 1 0 :: V2 CInt
        stepY = V2 0 1 :: V2 CInt
        w = Work2D 640 480
        n = 4 :: CInt

-- | Refresh a 'TextureObject' with images from a camera, and dump an
-- edge image into the given function.
textureCam :: (Vector Word8 -> IO ()) -> IO (IO TextureObject)
textureCam writeEdges =
  do [vid] <- genObjectNames 1
     textureBinding Texture2D $= Just vid
     texImage2D Nothing NoProxy 0 RGBA' (TextureSize2D 640 480) 0
                (PixelData RGB UnsignedByte nullPtr)
     textureFilter Texture2D $= ((Linear', Nothing), Linear')
     texture2DWrap $= (Repeated, Repeat)
     vidCam <- createCameraCapture (Just 0)
     let info = TexInfo 640 480 TexBGR
         refresh = do img <- vidCam :: IO ColorImage
                      let img' = canny 70 110 3 . convertBGRToGray $ img
                      withImagePixels img' writeEdges
                      reloadTexture vid . info $ packPixels img
     return (vid <$ refresh)
```

OpenCL

```haskell
let workSize = Work1D $ bufferLength verts
    sz = [640,480] :: [Int]
img <- allocImage [CL_MEM_READ_ONLY] sz :: CL (CLImage1 Float)
img' <- allocImage [CL_MEM_READ_WRITE] sz :: CL (CLImage1 Float)
let cleanup = () <$ (releaseObject verts >> releaseObject img)
k <- getKernel "etc/ripples.cl" "ripple"
blur <- mkBlur
return . (,cleanup) $ \e ->
  do writeImage img (V.map ((/255) . fromIntegral) e)
     blur img img'
     withGLObjects [bufferObject verts] $
       runKernelCL k img' verts workSize
```

```haskell
ripple :: BufferObject
ripple b = do verts <-
              let work
                  sz =
              img <- a
              img' <-
              let clea
              k <- get
              blur <-
              return .
                do wri
                   blu
                   wit
                     r

mkBlur :: CL (CLImage
mkBlur = do k <- getKe
            return $ \

  where stepX = V2 1 0
        stepY = V2 0 1
        w = Work2D 640 480
        n = 4 :: CInt

-- | Refresh a 'TextureObject' with images from a camera, and dump an
-- edge image into the given function.
textureCam :: (Vector Word8 -> IO ()) -> IO (IO TextureObject)
textureCam writeEdges =
  do [vid] <- genObjectNames 1
     textureBinding Texture2D $= Just vid
     texImage2D Nothing NoProxy 0 RGBA' (TextureSize2D 640 480) 0
                (PixelData RGB UnsignedByte nullPtr)
     textureFilter Texture2D $= ((Linear', Nothing), Linear')
     texture2DWrap $= (Repeated, Repeat)
     vidCam <- createCameraCapture (Just 0)
     let info = TexInfo 640 480 TexBGR
         refresh = do img <- vidCam :: IO ColorImage
                      let img' = canny 70 110 3 . convertBGRToGray $ img
                      withImagePixels img' writeEdges
                      reloadTexture vid . info $ packPixels img
     return (vid <$ refresh)
```

OpenCV

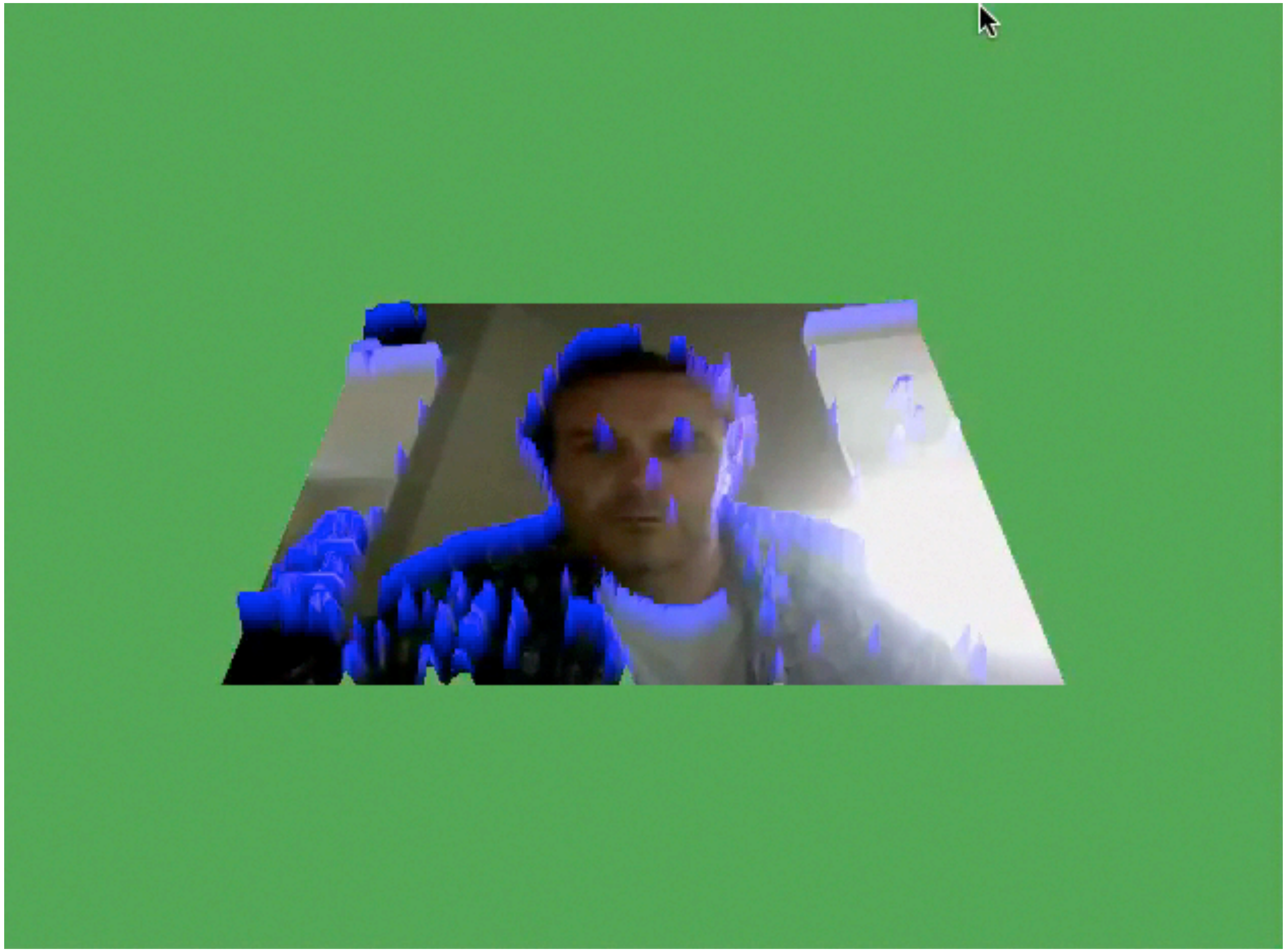
```haskell
ripple :: BufferObject -> CL (Vector Word8 -> CL (), IO ())
ripple b = do verts <- bufferFromGL b :: CL (CLBuffer (V4 Float))
              let workSize = Work1D $ bufferLength verts
                  sz = [640,480] :: [Int]
              img <- allocImage [CL_MEM_READ_ONLY] sz :: CL (CLImage1 Float)
              img' <- allocImage [CL_MEM_READ_WRITE] sz :: CL (CLImage1 Float)
              let cleanup = () <$ (releaseObject verts >> releaseObject img)
              k <- getKernel "etc/ripples.cl" "ripple"
              blur <- mkBlur
              return . (,cleanup) $ \e ->
                do writeImage img (V.map ((/255) . fromIntegral) e)
                   blur img img'
                   withGLObjects [bufferObject verts] $
                     runKernelCL k img' verts workSize

mkBlur :: CL (CLImage n a -> CLImage n a -> CL ())
mkBlur = do k <- getKernel "etc/ripples.cl" "localMax"
            return $ \i b -> do () <- runKernelCL k i b stepX n w
                                runKernelCL k i b stepY n w
  where stepX = V2 1 0 :: V2 CInt
        stepY = V2 0 1 :: V2 CInt
        w = Work2D 640 480
        n = 4 :: CInt

-- | Refresh a 'TextureObject' with images from a camera, and dump an
-- edge image into the given function.
textureCam :: (Vector Word8 -> IO ()) -> IO (IO TextureObject)
textureCam writeEdges =
  do [vid]
     textur
     texIma

     textur
     textur
     vidCam
     let in
         re

     return (vid <$ refresh)
```

```haskell
vidCam <- createCameraCapture (Just 0)
let info = TexInfo 640 480 TexBGR
    refresh = do img <- vidCam :: IO ColorImage
                 let img' = canny 70 110 3 . convertBGRToGray $ img
                 withImagePixels img' writeEdges
                 reloadTexture vid . info $ packPixels img
```

Final Tally

that’s a lot of functionality

109 LOC

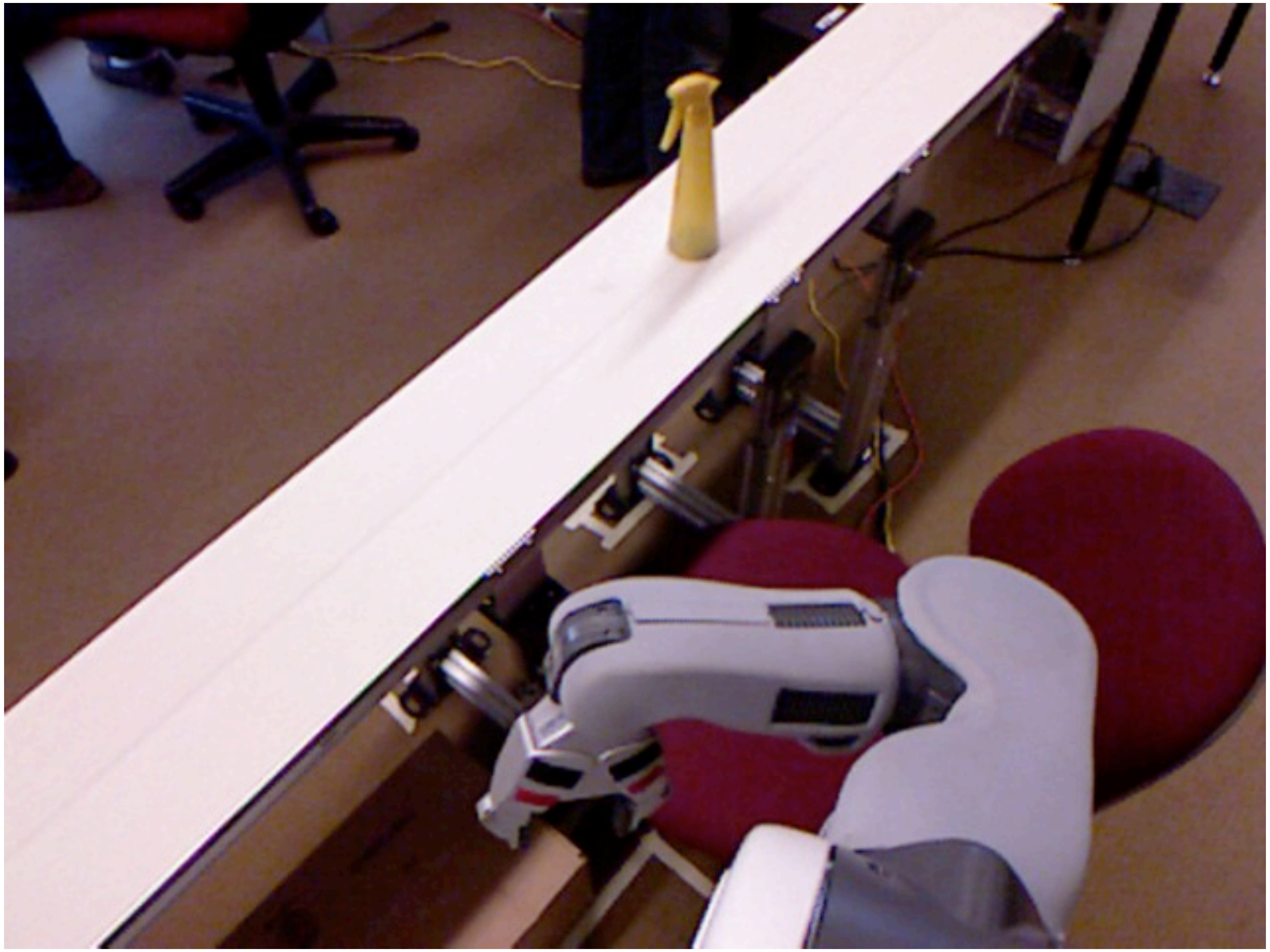


We Were Talking About Robots

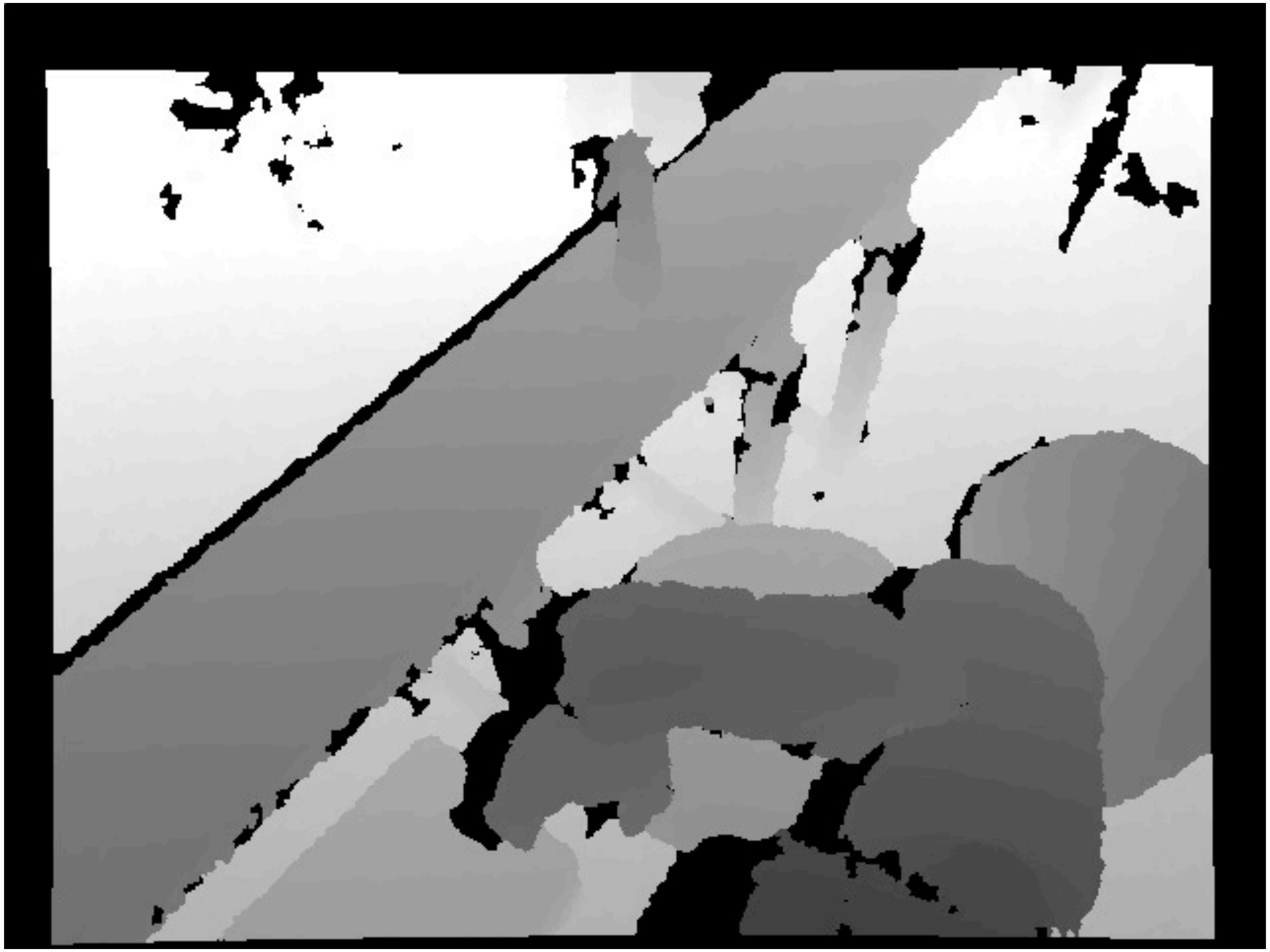
Fast Factory

Fast Factory

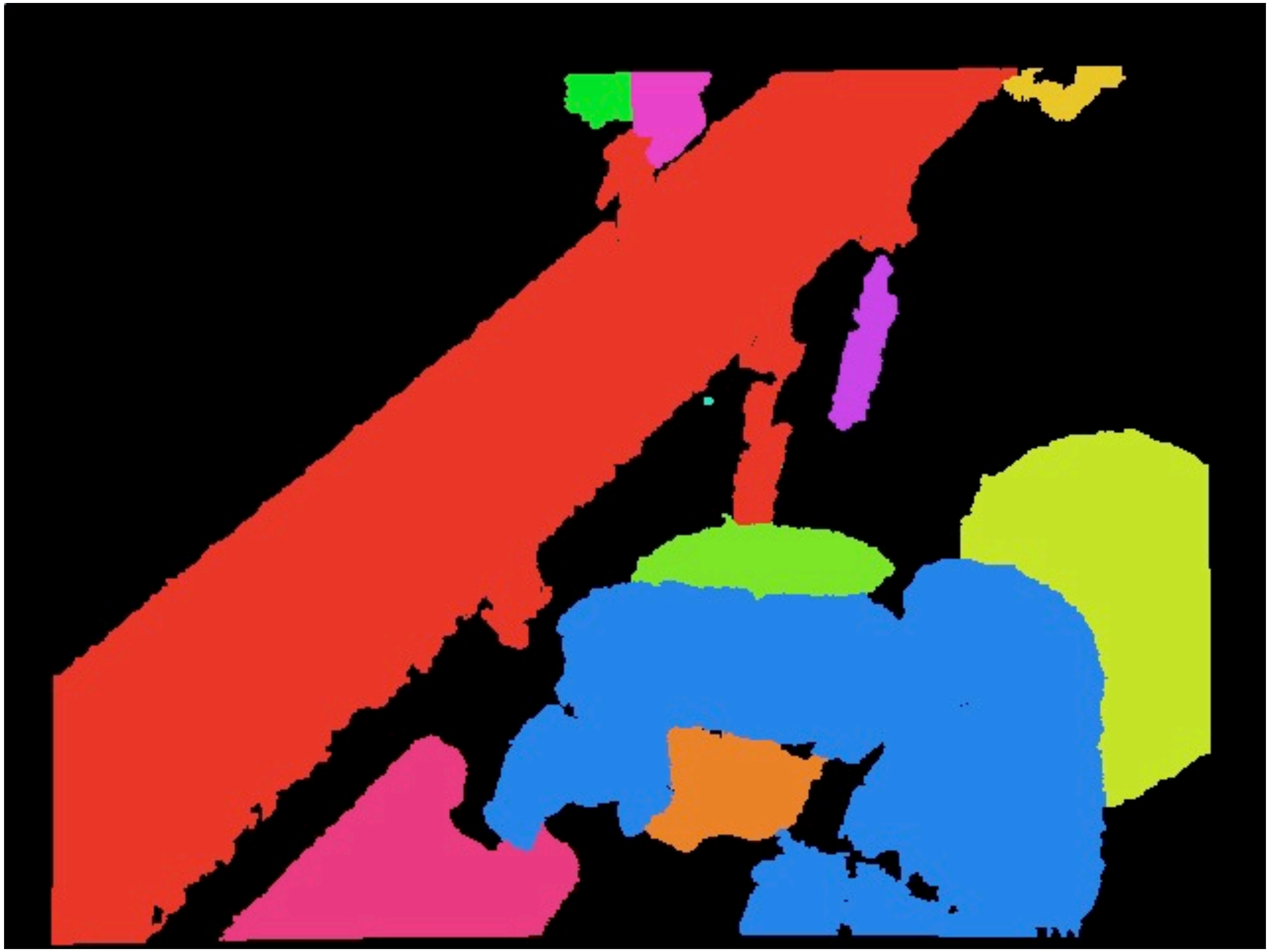
Robot's-Eye View



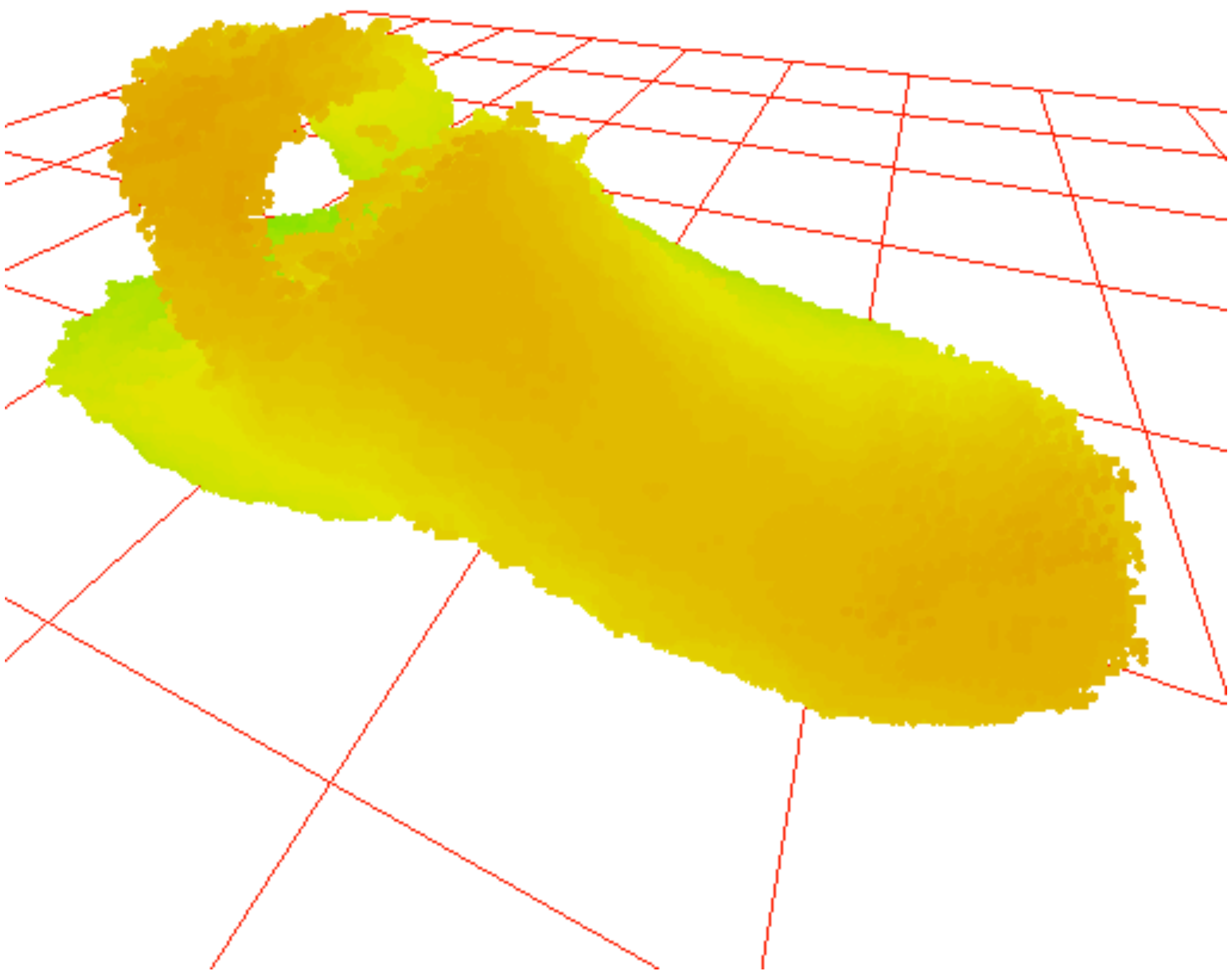
Seeing in 3D



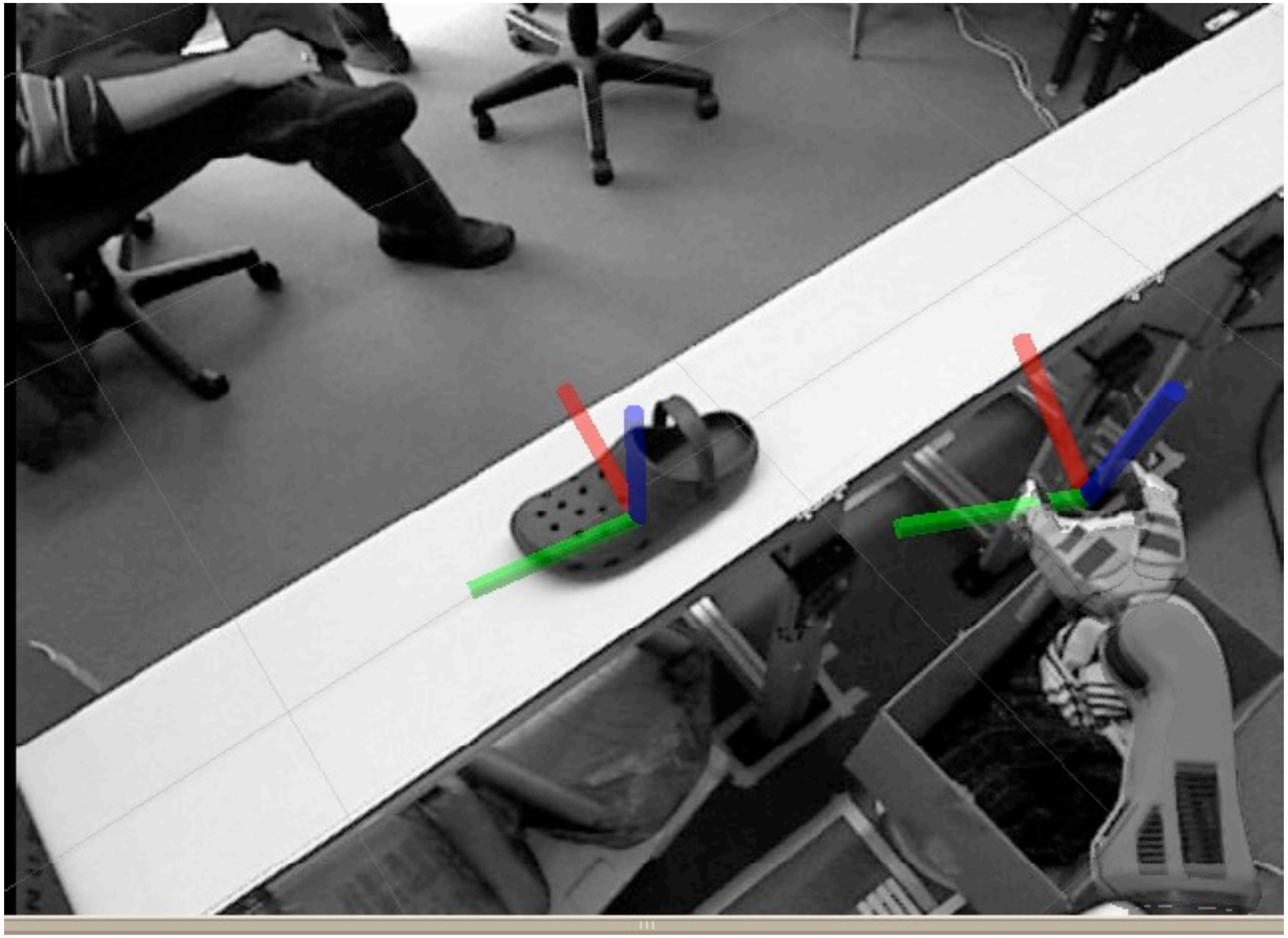
Parsing the Scene



3D Reference

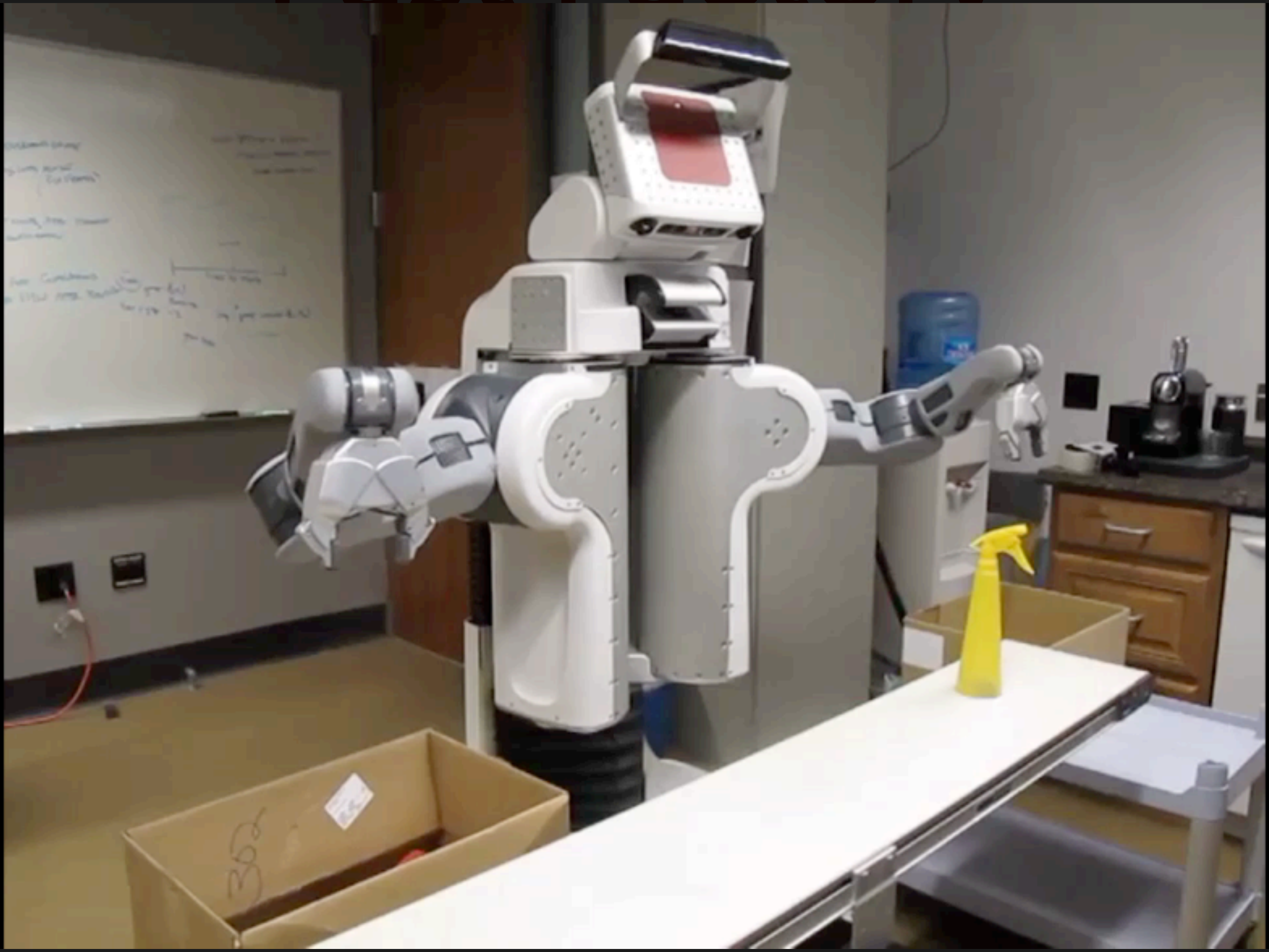


Coordinate Frames



Fast Factory

Fast Factory



Robotics Is...