```haskell
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE ApplicativeDo #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE MagicHash #-}
import System.Random
import System.Environment (getArgs)
import Debug.Trace
import Control.Applicative
import Data.List(sort, nub)
import Data.Proxy
import Control.Monad (replicateM)
import Data.Monoid hiding(Ap)
import Control.Monad
import Data.Bits
import GHC.Float
import GHC.Exts
import qualified Data.Map as M


compose :: Int -> (a -> a) -> (a -> a)
compose 0 f = id
compose n f = f . compose (n - 1) f

-- | Utility library for drawing sparklines

-- | List of characters that represent sparklines
sparkchars :: String
sparkchars = "_"

-- Convert an int to a sparkline character
num2spark :: RealFrac a => a -- ^ Max value
  -> a -- ^ Current value
  -> Char
num2spark maxv curv =
   sparkchars !!
     (floor $ (curv / maxv) * (fromIntegral (length sparkchars - 1)))

series2spark :: RealFrac a => [a] -> String
series2spark vs =
```

```haskell
  let maxv = if null vs then 0 else maximum vs
  in map (num2spark maxv) vs

seriesPrintSpark :: RealFrac a => [a] -> IO ()
seriesPrintSpark = putStrLn . series2spark

-- Probabilites
-- ============
type F = Float
-- | probablity density
type P = Float

-- | prob. distributions over space a
type D a = a -> P

-- | Scale the distribution by a float value
dscale :: D a -> Float -> D a
dscale d f a = f *  d a

uniform :: Int -> D a
uniform n _ = 1.0 / (fromIntegral $ n)


-- | Normal distribution with given mean
normalD :: Float -> (Float -> Float)
normalD mu f  =  exp (- ((f-mu)^2))

-- | Distribution that takes on value x^p for 1 <= x <= 2.  Is normalized
polyD :: Float -> (Float -> Float)
polyD p f = if 1 <= f && f <= 2 then (f ** p) * (p + 1) / (2 ** (p+1) - 1) else 0

type Random = Float
type Score = Float

-- | Trace all random choices made when generating this value
data Trace a = Trace { tval :: a, tscore :: Float, trs :: [Float] }




-- | Lift a pure value into a Trace value
mkTrace :: a -> Trace a
mkTrace a = Trace a 1.0 []

-- | multiply a score to a trace
scoreTrace :: Float -> Trace a -> Trace a
```

```haskell
scoreTrace f Trace{..} = Trace{tscore = tscore * f, ..}

prependRandomnessTrace :: Float -> Trace a -> Trace a
prependRandomnessTrace r Trace{..} = Trace { trs = r:trs, ..}

data PL x where
    Ret :: x -> PL x
    Sample01 :: (Float -> PL x) -> PL x
    Score :: Float -> PL x -> PL x
    Ap :: PL (a -> x) -> PL a -> PL x

instance Functor PL where
  fmap f (Ret x) = Ret (f x)
  fmap f (Sample01 r2plx) = Sample01 (\r -> fmap f (r2plx r))
  fmap f (Score s plx) = Score s (fmap f plx)
  fmap f (Ap pla2x pla) = Ap ((f .) <$> pla2x) pla


instance Applicative PL where
  pure = Ret
  pa2b <*> pa = Ap pa2b pa


instance Monad PL where
  return = Ret
  (Ret x) >>= x2ply = x2ply x
  (Sample01 r2plx) >>= x2ply = Sample01 (\r -> r2plx r >>= x2ply)
  (Score s plx) >>= x2ply = Score s (plx >>= x2ply)
  (Ap pla2x pla) >>= x2ply = pla2x >>= \a2x -> pla >>= \a -> x2ply (a2x a)

-- | operation to sample from [0, 1)
sample01 :: PL Float
sample01 = Sample01 Ret

score :: Float -> PL ()
score s = Score s (Ret ())

condition :: Bool -> PL ()
condition True = score 1
condition False = score 0

-- | convert a distribution into a PL
d2pl :: (Float, Float) -> D Float -> PL Float
d2pl (lo, hi) d = do
  u <- sample01
  let a = lo + u * (hi - lo)
```

```
  score $  d a
  return $ a

-- | A way to choose uniformly. Maybe slightly biased due to an off-by-one ;)
choose :: [a] -> PL a
choose as = do
    let l = length as
    u <- sample01
    let ix = floor $ u /  (1.0 / fromIntegral l)
    return $ as !! ix

instance MCMC a => MCMC (Trace a) where
  arbitrary = Trace { tval = arbitrary , tscore = 1.0, trs = []}
  uniform2val f = Trace { tval = uniform2val f,  tscore = 1.0, trs = []}

-- Typeclass that can provide me with data to run MCMC on it
class MCMC a where
    arbitrary :: a
    uniform2val :: Float -> a

instance MCMC Float where
    arbitrary = 0
    -- map [0, 1) -> (-infty, infty)
    uniform2val v = tan (-pi/2 + pi * v)


instance MCMC Int where
    arbitrary = 0
    -- map [0, 1) -> (-infty, infty)
    uniform2val v = floor $ tan (-pi/2 + pi * v)


-- | lift a regular computation into the Trace world, where we know what
-- decisions were taken.
reifyTrace :: PL x -> PL (Trace x)
reifyTrace (Ret x) = Ret (mkTrace x)
reifyTrace (Sample01 plx) = do
  r <- sample01
  trx <- reifyTrace $ plx r
  return $ prependRandomnessTrace r $ trx
reifyTrace (Score s plx) = do
  trx <- reifyTrace $ plx
  return $ scoreTrace s $ trx

-- | run the PL with the randomness provided, and then
-- return the rest of the proabilistic computation
```

```haskell
injectRandomness :: [Float] -> PL a -> PL a
injectRandomness _ (Ret x) = Ret x
injectRandomness (r:rs) (Sample01 r2plx)
 = injectRandomness rs (r2plx r)
injectRandomness [] (Sample01 r2plx) = (Sample01 r2plx)
injectRandomness rs (Score s plx) = Score s $ injectRandomness rs plx

-- | Replace the element of a list at a given index
replaceListAt :: Int -> a -> [a] -> [a]
replaceListAt ix a as = let (l, r) = (take (ix - 1) as, drop ix as)
                         in l ++ [a] ++ r


-- | Return a trace-adjusted MH computation
mhStepT_ :: PL (Trace x) -- ^ proposal
         -> Trace x -- ^ current position
         -> PL (Trace x)
mhStepT_ mtx tx = do
  -- | Return the original randomness, perturbed
  trs' <- do
      let l = length $ trs tx
      ix <- choose [0..(l-1)]
      r <- sample01
      return $ replaceListAt ix r (trs tx)
  -- | Run the original computation with the perturbation
  tx' <- injectRandomness trs' mtx
  let ratio = (tscore tx' * fromIntegral (length (trs tx'))) /
               (tscore tx * fromIntegral (length (trs tx)))
  r <- sample01
  return $ if r < ratio then tx' else tx

-- | Repeat monadic computation N times
repeatM :: Monad m => Int -> (a -> m a) -> (a -> m a)
repeatM 0 f x = return x
repeatM n f x = f x >>= repeatM (n - 1) f


-- | Transformer that adjusts a computation according to MH
mhT_ :: Trace x -> PL (Trace x) -> PL (Trace x)
mhT_ tx tmx = repeatM 10 (mhStepT_ tmx) $ tx

-- | Find a starting position that does not have probability 0
findNonZeroTrace :: PL (Trace x) -> PL (Trace x)
findNonZeroTrace mtx = do
  trx <- mtx
  if tscore trx /= 0
```

```haskell
  then return $ trx
  else findNonZeroTrace mtx


-- | run the computatation after taking weights into account
weighted :: MCMC x => PL x -> PL [x]
weighted mx =
  let mtx = reifyTrace mx
      go tx = do
        tx' <- mhT_ tx mtx
        liftA2 (:) (return tx) (go tx')
        -- txs <- go tx'
        -- return £ tx:txs
  in do
      tra <- findNonZeroTrace $ reifyTrace $ mx
      tras <- go tra -- Need Applicative instance here!
      return $ map tval tras

-- | Run the computation in an _unweighted_ fashion, not taking
-- scores into account
sample :: RandomGen g => g -> PL a -> (a, g)
sample g (Ret a) = (a, g)
sample g (Sample01 f2plnext) =
  let (f, g') = random g in sample g' (f2plnext f)
sample g (Score f plx) = sample g plx
sample g (Ap pla2x pla) =
  let (a2x, g1) = sample g pla2x
      (a, g2) = sample g1 pla
   in (a2x a, g2)



samples :: RandomGen g => Int -> g -> PL a -> ([a], g)
samples 0 g _ = ([], g)
samples n g pla = let (a, g') = sample g pla
                      (as, g'') = samples (n - 1) g' pla
                 in (a:as, g'')


-- | count fraction of times value occurs in list
occurFrac :: (Eq a) => [a] -> a -> Float
occurFrac as a =
    let noccur = length (filter (==a) as)
        n = length as
    in (fromIntegral noccur) / (fromIntegral n)
```

```haskell
-- | biased coin
coin :: Float -> PL Int -- 1 with prob. p1, 0 with prob. (1 - p1)
coin !p1 = do
    f <- sample01
    Ret $  if f <= p1 then 1 else 0

-- | fair dice
dice :: PL Int
dice = choose [1, 2, 3, 4, 5, 6]



-- | Create a histogram from values.
histogram :: Int -- ^ number of buckets
          -> [Float] -- values
          -> [Int]
histogram nbuckets as =
    let
        minv :: Float
        minv = minimum as
        maxv :: Float
        maxv = maximum as
        -- value per bucket
        perbucket :: Float
        perbucket = (maxv - minv) / (fromIntegral nbuckets)
        bucket :: Float -> Int
        bucket v = floor $ (v - minv) / perbucket
        bucketed :: M.Map Int Int
        bucketed = foldl (\m v -> M.insertWith (+) (bucket v) 1 m) mempty as
     in map snd . M.toList $ bucketed


printSamples :: (Real a, Eq a, Ord a, Show a) => String -> [a] -> IO ()
printSamples s as =  do
    putStrLn $ "***" <> s
    putStrLn $ "   samples: " <> series2spark (map toRational as)

printHistogam :: [Float] -> IO ()
printHistogam samples = putStrLn $ series2spark (map fromIntegral . histogram 10 $  samples)


-- | Given a coin bias, take samples and print bias
printCoin :: Float -> IO ()
printCoin bias = do
    let g = mkStdGen 1
    let (tosses, _) = samples 100 g (coin bias)
```

```
    printSamples ("bias: " <> show bias) tosses


-- | Create normal distribution as sum of uniform distributions.
normal :: PL Float
normal =  do
  xs <-(replicateM 1000 (coin 0.5))
  return $ fromIntegral (sum xs) / 500.0


-- | This file can be copy-pasted and will run!

-- | Symbols
type Sym = String
-- | Environments
type E a = M.Map Sym a
-- | Newtype to represent deriative values

newtype Der = Der { under :: F } deriving(Show, Num)

infixl 7 !#
-- | We are indexing the map at a "hash" (Sym)
(!#) :: E a -> Sym -> a
(!#) = (M.!)

-- | A node in the computation graph
data Node =
  Node { name :: Sym -- ^ Name of the node
       , ins :: [Node] -- ^ inputs to the node
       , out :: E F -> F -- ^ output of the node
       , der :: (E F, E (Sym -> Der))
                  -> Sym -> Der -- ^ derivative wrt to a name
       }

-- | @ looks like a "circle", which is a node. So we are indexing the map
-- at a node.
(!@) :: E a -> Node -> a
(!@) e node = e M.! (name node)

-- | Given the current environments of values and derivatives, compute
-- | The new value and derivative for a node.
run_ :: (E F, E (Sym -> Der)) -> Node -> (E F, E (Sym -> Der))
run_ ein (Node name ins out der) =
  let (e', ed') = foldl run_ ein ins -- run all the inputs
      v = out e' -- compute the output
      dv = der (e', ed') -- and the derivative
```

```haskell
  in (M.insert name v e', M.insert name dv ed')  -- and insert them

-- | Run the program given a node
run :: E F -> Node -> (E F, E (Sym -> Der))
run e n = run_ (e, mempty) n

-- | Let's build nodes
nconst :: Sym -> F -> Node
nconst n f = Node n [] (\_ -> f) (\_ _ -> 0)

-- | Variable
nvar :: Sym -> Node
nvar n = Node n [] (!# n) (\_ n' -> if n == n' then 1 else 0)

-- | binary operation
nbinop :: (F -> F -> F)  -- ^ output computation from inputs
 -> (F -> Der -> F -> Der -> Der) -- ^ derivative computation from outputs
 -> Sym -- ^ Name
 -> (Node, Node) -- ^ input nodes
 -> Node
nbinop f df n (in1, in2) =
  Node { name = n
       , ins = [in1, in2]
       , out = \e -> f (e !# name in1) (e !# name in2)
       , der = \(e, ed) n' ->
                 let (name1, name2) = (name in1, name in2)
                     (v1, v2) = (e !# name1, e !# name2)
                     (dv1, dv2) = (ed !# name1 $ n', ed !# name2 $ n')
                     in df v1 dv1 v2 dv2
       }

nadd :: Sym -> (Node, Node) -> Node
nadd = nbinop (+) (\v dv v' dv' -> dv + dv')

nmul :: Sym -> (Node, Node) -> Node
nmul = nbinop (*) (\v (Der dv) v' (Der dv') -> Der $ (v*dv') + (v'*dv))

-- | 3 vector
data Vec3 = Vec3 { vx :: Float, vy :: Float, vz :: Float }

instance Semigroup Vec3 where
  (<>) = (^+)
instance Monoid Vec3 where
  mempty = zzz
  mappend = (<>)
```

```haskell
-- | get maximum component
vmax :: Vec3 -> Float
vmax (Vec3 vx vy vz) = foldl1 max [vx, vy, vz]

-- | vector addition
(^+) :: Vec3  -> Vec3 -> Vec3
(^+) (Vec3 x y z) (Vec3 x' y' z') =
  Vec3 (x + x') (y + y') (z + z')

-- | vector subtraction
(^-) :: Vec3 -> Vec3 -> Vec3
(^-) x y = x ^+ ((-1.0) ^* y)

-- | sclar multiplication
(^*) :: Float -> Vec3 -> Vec3
(^*) r (Vec3 x y z) =
  Vec3 (x * r) (y * r) (z * r)

(^/) :: Vec3 -> Float -> Vec3
v ^/ r = (1.0 / r) ^* v

-- | dot product
(^.) :: Vec3 -> Vec3 -> Float
(^.) (Vec3 x y z) (Vec3 x' y' z') = (x * x') + (y * y') + (z * z')

veclensq :: Vec3 -> Float
veclensq v = v ^. v

veclen :: Vec3 -> Float
veclen = sqrt . veclensq

cosine :: Vec3 -> Vec3 -> Float
cosine v w = v ^. w / ((veclen v) * (veclen w))

-- | cross product
cross :: Vec3 -> Vec3 -> Vec3
cross (Vec3 x y z) (Vec3 x' y' z') =
  let xnew = y * z' - z * y'
      ynew = z * x' - x * z'
      znew = x * y' - y * x'
   in Vec3 xnew ynew znew

vecnorm :: Vec3 -> Vec3
vecnorm v =  (1.0 / veclen v) ^*  v
```

```haskell
-- | zero vector
zzz :: Vec3
zzz = Vec3 0.0 0.0 0.0

xzz :: Vec3
xzz = Vec3 1.0 0.0 0.0

zyz :: Vec3
zyz = Vec3 0.0 1.0 0.0

--  | ray with origin and direction
data Ray = Ray { rorigin :: Vec3, rdir :: Vec3}

-- | project the ray for some magnitude
(-->) :: Ray -> Float -> Vec3
Ray{..} --> d = rorigin ^+ (d ^* rdir)

data Refl = Diff | Specular | Refract

data Sphere =
  Sphere { srad :: Float
         , spos :: Vec3
         , semission :: Vec3
         , scolor :: Vec3
         , srefl :: Refl
         }


-- | Get the normal vector from the center of a sphere to a point
sphereNormal :: Sphere -> Vec3 -> Vec3
sphereNormal Sphere{..} pos =
  vecnorm $ pos ^- spos

-- | List of spheres to render
gspheres :: [Sphere]
gspheres =
  --[ Sphere 0.2 (Vec3 0.0 0.0 (-2.0)) (Vec3 1.0 1.0 1.0) (Vec3 1.0 1.0 1.0) Diff,
  [ -- Sphere 0.8 (Vec3 0.0 (-0.5) 3.0) zzz (Vec3 0 1 0) Refract,
    -- Sphere 0.2 (Vec3 (-0.3) 0.0 2.0) zzz (Vec3 1.0 0.0 0.0) Diff,
    -- Sphere 0.2 (Vec3 0.3 0.0 2.0) zzz (Vec3 0.0 0.0 1.0) Diff,
    -- Sphere 0.2 (Vec3 0.0 0.0 1.5) zzz (Vec3 1.0 1.0 0.0) Refract,
    Sphere 5000000 (Vec3 (-5000000-20) 0 0) (Vec3 0 0 1) zzz Diff, -- left
    Sphere 5000000 (Vec3 (5000000+20) 0 0) (Vec3 1 0 0) zzz Diff, -- right
    Sphere 5000000 (Vec3 0 0 (5000000+99)) (Vec3 0 1 0) zzz Diff, -- back
    Sphere 5000000 (Vec3 0 (5000000+10) 0) (Vec3 1 1 0) zzz Diff, -- bottom
    Sphere 5000000 (Vec3 0 (-5000000-10) 0) (Vec3 0 1 1) zzz Diff, -- top
```

```
    Sphere 40 (Vec3 0 (-48) 50) (Vec3 1 1 1) zzz Diff -- light
  ]

-- | epsilon
eps :: Float
eps = 0.0001

-- | solve quadratic and return the smaller root
solveQuadratic :: Float -> Float -> Float -> [Float]
solveQuadratic a b c =
  let disc = b*b - 4*a*c
   in if disc < 0
      then []
      else let r = (-b + sqrt disc) / (2 * a)
               r' = (-b - sqrt disc) / (2 * a)
            in [r, r']

-- |x - spos|^2 = srad^2
-- x = rorigin + t . rdir
-- | we assume that the ray direction is *normalized*
sintersect :: Ray -> Sphere -> Maybe Float
sintersect Ray{..} Sphere{..} = do
  let o = spos ^- rorigin  -- ^ original relative to ray corrdiates
      a = rdir ^. rdir
      b = -2.0 * (rdir ^. o)
      c = o ^. o - srad * srad
      roots = [r | r <- solveQuadratic a b c, r >= 0]
   in case roots of
        [] -> Nothing
        [r] -> Just r
        [r, r'] ->  Just $ min r r'


-- | Return the smallest value from a list
listmin :: (Ord o) => (a -> Maybe o) -> [a] -> Maybe (a, o)
listmin f [] = Nothing
listmin f (x:xs) =
  case (listmin f xs, f x) of
    (other, Nothing) -> other
    (Nothing, Just xcmp) -> Just (x, xcmp)
    (Just (x', x'cmp), Just xcmp) ->
          pure $ if xcmp < x'cmp then (x, xcmp) else (x', x'cmp)

-- | Get the closest sphere along a ray and the distance traveled
closestSphere :: Ray ->  Maybe (Sphere, Float)
closestSphere r = listmin (sintersect r) gspheres
```

```haskell
clamp01 :: Float -> Float
clamp01 f
  | f < 0 = 0
  | f > 1 = 1
  | otherwise = f

vclamp01 :: Vec3 -> Vec3
vclamp01 (Vec3 x y z) = Vec3 (clamp01 x) (clamp01 y) (clamp01 z)

-- | Return the color of the surface of the sphere at this
-- angle of the viewing ray, given the point of contact
surfaceColor :: Ray -> Sphere -> Vec3 -> Vec3
surfaceColor r s hitpoint = let factor = abs (cosine (rdir r) (sphereNormal s hitpoint))
 in factor ^* (scolor s)



-- | return a random ray in a hemisphere at a position
randRayAt :: Vec3 -- ^ position
          -> Vec3 -- ^ hemisphere normal
          -> PL Ray
randRayAt p n = do
  -- | angle to the normal vector
  thetaToNormal <- (0.5 * pi *) <$> sample01
  -- | pick a uniform angle on the circle picked by the theta to normal
  thetaCircle <- (2.0 * pi *) <$> sample01
  -- | right now, I'm going to fuck around and implement something somewhat incorrect
  -- apply some small random perturbation to the given normal vector...
  r1 <- (\x -> (x - 0.5)*0.05) <$> sample01
  r2 <- (\x -> (x - 0.5)*0.05) <$> sample01
  let x' = vx n + r1
  let y' = vx n + r2
  let z' = sqrt (1.0 - x'*x' - y'*y')
  -- | move the origin along the normal so it doesn't intersect the sphere again...
  return $ Ray (p ^+ (0.01 ^* n)) n

-- | Given colors and the viewing angle, get the final color
mergeLightColors :: Vec3 -> Vec3 -> [Vec3] -> Vec3
mergeLightColors view hitpoint vs =
  vclamp01 $  foldl (^+) zzz vs

v3map :: (Float -> Float) -> Vec3 -> Vec3
v3map f (Vec3 x y z) = Vec3 (f x) (f y) (f z)
colormul :: Vec3 -> Vec3 -> Vec3
```

```haskell
colormul (Vec3 x y z) (Vec3 x' y' z') = Vec3 (x*x') (y*y') (z*z')

-- | take average of vectors
vecavg :: [Vec3] -> Vec3
vecavg [] = mempty
vecavg vs = mconcat vs ^/ (fromIntegral $ length vs)

-- | NOTE: assumes the vector we are projecting on is normalized
vecprojecton :: Vec3 -- ^ vector to be projected
             -> Vec3 -- ^ subspace on which we are projecting
             -> Vec3
vecprojecton v vp = let vpnorm = vecnorm vp in (v ^. vpnorm) ^* vpnorm

-- | find the rejection of the vector along this diretion
vecrejecton :: Vec3 -> Vec3 -> Vec3
vecrejecton v vp = v ^- vecprojecton v vp

-- | reflect the vector about another vector
vecReflect :: Vec3 -> Vec3 -> Vec3
vecReflect v n = vecprojecton v n ^- vecrejecton v n

-- | ramp the value, by creating "hard steps"
ramp :: Int -> Float -> Float
ramp i f = (fromIntegral (floor (f * fromIntegral i))) / (fromIntegral i)

-- https://www.cs.cmu.edu/afs/cs/academic/class/15462-f09/www/lec/lec8.pdf
-- https://maverick.inria.fr/~Nicolas.Holzschuch/cours/Slides/1b_Materiaux.pdf
-- http://www.graphics.stanford.edu/courses/cs348b-01/course29.hanrahan.pdf
-- | path trace
mcpt :: (Ray, Float) -- ^ given ray and weight of ray
     -> Int -- ^ Given depth of number of bounces
     -> PL Vec3 -- ^ return final color
mcpt (ray, w) 4 = return $ zzz
mcpt (ray, w) depth = do
  case closestSphere ray of
    Nothing -> do
      score 0.1 -- we want to _avoid_ this region of program space!
      return $  zzz
    Just (sphere@Sphere{srefl=Refract}, raylen) -> do
        let hitpoint = ray --> raylen
        let normal = sphereNormal sphere hitpoint
        let project = vecprojecton (rdir ray) normal
        let reject = vecrejecton (rdir ray) normal

        let refracted = (1.4 ^* project) ^+ reject
        let rayReflected = Ray (hitpoint ^+ (0.01 ^* normal)) (vecReflect ((-1.0) ^* (rdir r
```

```
        let rayRefracted = Ray (hitpoint ^+ (0.01 ^* refracted)) (vecnorm $ refracted)
        refracted <- mcpt (rayRefracted, w) (depth + 1)
        reflected <- mcpt (rayReflected, w) (depth + 1)
        return $ v3map (ramp 4) $ (0.2 ^* reflected) ^+ (0.8 ^* refracted)

    Just (sphere@Sphere{srefl=Specular}, raylen) -> do
        let hitpoint = ray --> raylen
        let normal = sphereNormal sphere hitpoint
        let rayReflected = Ray (hitpoint ^+ (0.01 ^* normal)) (vecReflect ((-1.0) ^* (rdir r
        return  $ error $ "unimplemented"

    Just (sphere@Sphere{srefl=Diff}, raylen) -> do
        let hitpoint = ray --> raylen
        let normal = sphereNormal sphere hitpoint
        -- | ray going out
        let rayReflected = Ray (hitpoint ^+ (0.01 ^* normal)) (vecReflect ((-1.0) ^* (rdir r

        -- | local diffuse color
        incomingrays <- replicateM 1 $ do
                  -- rayOutward <- -- randRayAt hitpoint normal
                  let rayOutward = rayReflected
                  color <- mcpt (rayOutward, w) (depth + 1)
                  return $ (rayOutward, color)
        let incomingColor = vecavg $ [ (clamp01 $ cosine (rdir rayOutward) normal) ^* lightc
        let localDiffuse = colormul (scolor sphere) incomingColor
        return $ (semission sphere) ^+ (v3map (ramp 5) $ localDiffuse) -- localEmission --

-- | A distribution over coin biases, given the data of the coin
-- flips seen so far. 1 or 0
-- TODO: Think of using CPS to
-- make you be able to scoreDistribution the distribution
-- you are sampling from!
predictCoinBias :: [Int] -> PL [Float]
predictCoinBias flips = weighted $ do
  b <- sample01
  forM_ flips $ \f -> do
    score $ if f == 1 then b else (1 - b)
  return $ b



main :: IO ()
main = do
    args <- getArgs
    case args !! 1  of
```

```
    "foo" -> putStrLn $ "foo"
    "bar" -> putStrLn $ "bar"
    _ -> putStrLn $ "unknown"

-- let g = mkStdGen 1
-- printCoin 0.1
-- printCoin 0.8
-- printCoin 0.5
-- printCoin 0.7

-- let (mcmcsamples, _) = samples 10 g (dice)
-- printSamples "fair dice" (fromIntegral <£> mcmcsamples)


-- putStrLn £ "biased dice : (x == 1 || x == 6)"
-- let (mcmcsamples, _) =
--       sample g
--         (weighted £ (do
--                 x <- dice
--                 condition (x <= 1 || x >= 6)
--                 return x))
-- putStrLn £ "biased dice samples: " <> (show £ take 10 mcmcsamples)
-- printSamples "bised dice: " (fromIntegral <£> take 100 mcmcsamples)

-- putStrLn £ "normal distribution using central limit theorem: "
-- let (nsamples, _) = samples 1000 g normal
-- -- printSamples "normal: " nsamples
-- printHistogam £  nsamples


-- putStrLn £ "normal distribution using MCMC: "
-- let (mcmcsamples, _) = sample g (weighted £ d2pl (-10, 10) £ normalD 0.5)
-- printHistogam £ take 10000 £  mcmcsamples

-- putStrLn £ "sampling from x^4 with finite support"
-- let (mcmcsamples, _) = sample g (weighted £ d2pl (0, 5)£  \x -> x ** 2)
-- printHistogam £ take 1000  mcmcsamples


-- putStrLn £ "sampling from |sin(x)| with finite support"
-- let (mcmcsamples, _) = sample g (weighted £ d2pl (0, 6)£  \x -> abs (sin x))
-- printHistogam £ take 10000 mcmcsamples


-- putStrLn £ "bias distribution with supplied with []"
-- let (mcmcsamples, _) = sample g (predictCoinBias [])
```

```
-- printHistogam £ take 1000 £ mcmcsamples

-- putStrLn £ "bias distribution with supplied with [True]"
-- let (mcmcsamples, _) = sample g (predictCoinBias [1, 1])
-- printHistogam £ take 1000 £ mcmcsamples


-- putStrLn £ "bias distribution with supplied with [0] x 10"
-- let (mcmcsamples, _) = sample g (predictCoinBias (replicate 10 0))
-- printHistogam £ take 100 £ mcmcsamples

-- putStrLn £ "bias distribution with supplied with [1] x 2"
-- let (mcmcsamples, _) = sample g (predictCoinBias (replicate 2 1))
-- printHistogam £ take 100 £ mcmcsamples

-- putStrLn £ "bias distribution with supplied with [1] x 30"
-- let (mcmcsamples, _) = sample g (predictCoinBias (replicate 30 1))
-- printHistogam £ take 100 £ mcmcsamples


-- putStrLn £ "bias distribution with supplied with [0, 1]"
-- let (mcmcsamples, _) = sample g (predictCoinBias (mconcat £ replicate 10 [0, 1]))
-- printHistogam £ take 100 £ mcmcsamples


-- putStrLn £ "bias distribution with supplied with [1, 0]"
-- let (mcmcsamples, _) = sample g (predictCoinBias (mconcat £ replicate 20 [1, 0]))
-- printHistogam £ take 100 £ mcmcsamples
```