

Optimizing smallpt

Davean Scies, Siddharth Bhat

Haskell Exchange

November 4th, 2020

What is smallpt anyway?

What is smallpt anyway?



- ▶ 99 LoC C++ raytracer.
- ▶ Perfect for an optimization case study.
- ▶ Ported to many languages, including Haskell! (Thanks to Vo Minh Thu([noteed](#))).
- ▶ Start from [noteed](#)'s original source; SHA the output image for baseline, keep optimizing.
- ▶ Plan: Quick walk through Haskell code, end up at C++ (`clang++`) performance.

What is smallpt anyway?

```
struct Vec {
    double x, y, z; // position, also color (r,g,b)
    ... methods...
};

struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };

enum Refl_t { DIFF, SPEC, REFR }; // material types, used in radiance()

struct Sphere {
    double rad; // radius
    Vec p, e, c; // position, emission, color
    Refl_t refl; // reflection type (DIFFuse, SPECular, REFRactive)
    ... methods ...
    double intersect(const Ray &r) const // returns distance, 0 if nohit
};

Sphere spheres[] = {//Scene: radius, position, emission, color, material
    Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
    ... initialization ...
};
```

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
```

```
}
```

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
```

```
    radiance
```

```
    radiance
```

```
    radiance
```

```
    radiance  
    radiance
```

```
    radiance  
    radiance
```

```
}
```

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
```

```
    if (          ) if (          )          else  
    if (          ){
```

```
        radiance  
    } else if (          )  
        radiance
```

```
    if (          )  
        radiance
```

```
        radiance          radiance  
        radiance          radiance  
    }
```

What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
```

```
Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
```

```
if (          ) if (          )          else  
if (          ){
```

```
          radiance  
} else if (          )  
          radiance
```

```
if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)  
    radiance
```

```
    radiance          radiance  
    radiance          radiance  
}
```


What is smallpt anyway?

```
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
    double t; // distance to intersection
    int id=0; // id of intersected object
    if (!intersect(r, t, id)) return Vec(); // if miss, return black
    const Sphere &obj = spheres[id]; // the hit object
    Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
    double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
    if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
    if (obj.refl == DIFF){ // Ideal DIFFUSE reflection
        double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
        Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
        Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
        return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
    } else if (obj.refl == SPEC) // Ideal SPECULAR reflection
        return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
    Ray reflRay(x, r.d-n*2*n.dot(r.d)); // Ideal dielectric REFRACTION
    bool into = n.dot(nl)>0; // Ray from outside going in?
    double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
    if ((cos2t=1-nnt*nnt*(1-ddn*ddn)<0) // Total internal reflection
        return obj.e + f.mult(radiance(reflRay,depth,Xi));
    Vec tdir = (r.d*nnt - n*((into?-1):1)*(ddn*nnt+sqrt(cos2t))).norm();
    double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
    double Re=R0+(1-R0)*c*c*c*c*c, Tr=1-Re, P=.25+.5*Re, RP=Re/P, TP=Tr/(1-P);
    return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ? // Russian roulette
        radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
        radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
```

Initial Haskell Code: radiance (1×)

```
radiance :: Ray -> CInt -> Ptr CUSHort -> IO Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
  (Nothing,_) -> return zerov
  (Just t,Sphere _r p e c refl) -> do
```

```
    continue f = case refl of
      DIFF -> do
```

```
        radiance
```

```
      SPEC -> do
```

```
        rad <- radiance
```

```
      REFR -> do
```

```
        if
```

```
        then do
```

```
          rad <- radiance reflRay depth' xi
```

Initial Haskell Code: radiance (1×)

```
radiance :: Ray -> CInt -> Ptr CUSHort -> IO Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
  (Nothing,_) -> return zeroV
  (Just t,Sphere _r p e c refl) -> do
    let x = o `addv` (d `mulvs` t)
        n = norm $ x `subv` p
        nl = if n `dot` d < 0 then n else n `mulvs` (-1)
        pr = maxv c
        depth' = depth + 1
    continue f = case refl of
      DIFF -> do
        r1 <- ((2*pi)* `fmap` erand48 xi
        r2 <- erand48 xi
        let r2s = sqrt r2
            w@(Vec wx _ _) = nl
            u = norm $ (if abs wx > 0.1 then (Vec 0 1 0) else (Vec 1 0 0)) `cross` w
            v = w `cross` u
            d' = norm $ (u`mulvs`(cos r1*r2s)) `addv` (v`mulvs`(sin r1*r2s)) `addv` (w`mulvs`sqrt (1-r2))
        rad <- radiance (Ray x d') depth' xi
        return $ e `addv` (f `mulv` rad)
      SPEC -> do
        let d' = d `subv` (n `mulvs` (2 * (n`dot`d)))
        rad <- radiance (Ray x d') depth' xi
        return $ e `addv` (f `mulv` rad)
      REFR -> do
        let reflRay = Ray x (d `subv` (n `mulvs` (2* n`dot`d))) -- Ideal dielectric REFRACTION
            into = n`dot`nl > 0 -- Ray from outside going in?
            nc = 1
            nt = 1.5
            nnt = if into then nc/nt else nt/nc
            ddn = d`dot`nl
            cos2t = 1-nnt*nnt*(1-ddn*ddn)
        if cos2t<0 -- Total internal reflection
        then do
          rad <- radiance reflRay depth' xi
```

Initial Haskell Code: Entry point (1×)

```
smallpt :: Int -> Int -> Int -> IO ()
smallpt w h nsamps = do
    ...
    c <- VM.replicate (w * h) 0
    allocaArray 3 \xi -> -- Create mutable memory
    flip mapM_ [0..h-1] $ \y -> do -- Loop
        writeXi xi y
        for_ [0..w-1] \x -> do -- Loop
            let i = (h-y-1) * w + x
            for_ [0..1] \sy -> do -- Loop
                for_ [0..1] \sx -> do -- Loop
                    r <- newIORef 0 -- Create mutable memory
                    for_ [0..samps-1] \s -> do -- Loops, Loops
                        r1 <- (2*) <$> erand48 xi
                        ...
                        rad <- radiance (Ray (org+d.*140) (norm d)) 0 xi -- Crunch
                        ...
                        modifyIORef r (+ rad .* recip (fromIntegral samps)) -- Write
                    ci <- VM.unsafeRead c i
                    Vec rr rg rb <- readIORef r
                    VM.unsafeWrite c i $
                        ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25 -- Write
```

Initial Haskell Code: File I/O (1×)

```
withFile "image.ppm" WriteMode $ \hdl -> do
  hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
  flip mapM_ [0..w*h-1] \i -> do
    Vec r g b <- VM.unsafeRead c i
    hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
```

Initial Haskell Code: RNG (1×)

```
foreign import ccall unsafe "erand48"  
  erand48 :: Ptr CUSHort -> IO Double
```

Restrict export list to 'main' (1.13×)

```
-module Main where  
+module Main (main) where
```

Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
-data Ray = Ray Vec Vec -- origin, direction
+data Ray = Ray {-# UNPACK #-} !Vec {-# UNPACK #-} !Vec -- origin, direction

data Refl = DIFF | SPEC | REFR -- material types, used in radiance

-- radius, position, emission, color, reflection
-data Sphere = Sphere Double Vec Vec Vec !Refl
+data Sphere = Sphere {-# UNPACK #-} !Double
+                      {-# UNPACK #-} !Vec
+                      {-# UNPACK #-} !Vec
+                      {-# UNPACK #-} !Vec !Refl
```


Mark entries of Ray and Sphere as UNPACK and Strict (1.07×)

```
-data Ray = Ray Vec Vec -- origin, direction
+data Ray = Ray {-# UNPACK #-} !Vec {-# UNPACK #-} !Vec -- origin, direction

data Refl = DIFF | SPEC | REFR -- material types, used in radiance

-- radius, position, emission, color, reflection
-data Sphere = Sphere Double Vec Vec Vec !Refl
+data Sphere = Sphere {-# UNPACK #-} !Double
+                      {-# UNPACK #-} !Vec
+                      {-# UNPACK #-} !Vec
+                      {-# UNPACK #-} !Vec !Refl
```

Use a pattern synonym to unpack Refl in Sphere (1.07×)

```
+{-# LANGUAGE PatternSynonyms #-}

-data Refl = DIFF | SPEC | REFR -- material types, used in radiance
+newtype Refl = Refl Int -- material types, used in radiance
+pattern DIFF,SPEC,REFR :: Refl
+pattern DIFF = Refl 0
+pattern SPEC = Refl 1
+pattern REFR = Refl 2
+{-# COMPLETE DIFF, SPEC, REFR #-}

-- radius, position, emission, color, reflection
data Sphere = Sphere {-# UNPACK #-} !Double
                    {-# UNPACK #-} !Vec
                    {-# UNPACK #-} !Vec
-                    {-# UNPACK #-} !Vec !Refl
+                    {-# UNPACK #-} !Vec {-# UNPACK #-} !Refl
```

Change from maximum on a list to max (1.08x)

```
-maxv (Vec a b c) = maximum [a,b,c]
+maxv (Vec a b c) = max a (max b c)

    let x = o `addv` (d `mulvs` t)
        n = norm $ x `subv` p
        nl = if n `dot` d < 0 then n else n `mulvs` (-1)
-      pr = maxv c
    depth' = depth + 1
    continue f = case refl of
        DIFF -> do
...
    if depth'>5
    then do
        er <- erand48 xi
+      let !pr = maxv c
```

Convert erand48 to pure Haskell (1.09×)

```
-radiance :: Ray -> CInt -> Ptr CUSHort -> IO Vec
+radiance :: Ray -> Int -> IORef Word64 -> IO Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
  (Nothing,_) -> return zerov
  (Just t,Sphere _r p e c refl) -> do
@@ -153,9 +153,8 @@ smallpt w h nsamps = do
    cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
    cy = norm (cx `cross` dir) `mulvs` 0.5135
    c <- VM.replicate (w * h) zerov
-   allocaArray 3 $ \xi ->
-     flip mapM_ [0..h-1] $ \y -> do
+   xi <- newIORef 0
+   flip mapM_ [0..h-1] $ \y -> do
      writeXi xi y
```

Remove mutability: Erand48 Monad

```
-erand48 :: IORef Word64 -> IO Double
-erand48 !t = do
-  r <- readIORef t
+data ET a = ET !Word64 !a deriving Functor
+newtype Erand48 a = Erand48 { runErand48' :: Word64 -> ET a } deriving Functor
+instance Applicative Erand48 where
+instance Monad Erand48 where
+runWithErand48 :: Int -> Erand48 a -> a
+erand48 :: Erand48 Double
...
-radiance :: Ray -> Int -> IORef Word64 -> IO Vec
-radiance ray@(Ray o d) depth xi = case intersects ray of
+radiance :: Ray -> Int -> Erand48 Vec
+radiance ray@(Ray o d) depth = case intersects ray of
...
-      r1 <- (2*pi*) <$> erand48 xi
-      r2 <- erand48 xi
+      r1 <- (2*pi*) <$> erand48
+      r2 <- erand48
...
-      then (* rp) <$> radiance reflRay depth' xi
-      else (* tp) <$> radiance (Ray x tdir) depth' xi
+      then (* rp) <$> radiance reflRay depth'
+      else (* tp) <$> radiance (Ray x tdir) depth'
```

Removing mutation: eliminate IORef

```
- c <- VM.replicate (w * h) 0
- xi <- newIORef 0
- flip mapM_ [0..h-1] $ \y -> do
-   writeXi xi y
-   for_ [0..w-1] \x -> do
-     let i = (h-y-1) * w + x
-     for_ [0..1] \sy -> do
-       for_ [0..1] \sx -> do
-         r <- newIORef 0
-         for_ [0..samps-1] \_s -> do
-           r1 <- (2*) <$> erand48 xi
+ img = (`concatMap` [(h-1),(h-2)..0]) $ \y -> runWithErand48 y do
+   for [0..w-1] \x -> do
+     (\pf -> foldlM pf 0 [(sy, sx) | sy <- [0,1], sx <- [0,1]]) \ci (sy, sx) -> do
+       Vec rr rg rb <- (\f -> foldlM f 0 [0..samps-1]) \!r _s -> do
+         r1 <- (2*) <$> erand48
+
+ ...
-       modifyIORef r (+ rad .* recip (fromIntegral samps))
-       ci <- VM.unsafeRead c i
-       Vec rr rg rb <- readIORef r
-       VM.unsafeWrite c i $ ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
+       pure (r + rad .* recip (fromIntegral samps))
+       pure (ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25)
+
+ ...
```

Set everything in smallpt to be strict (1.17×)

```
smallpt :: Int -> Int -> Int -> IO ()
smallpt w h nsamps = do
-   let samps = nsamps `div` 4
-   org = Vec 50 52 295.6
-   dir = norm $ Vec 0 (-0.042612) (-1)
-   cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
-   cy = norm (cx `cross` dir) `mulvs` 0.5135
+   let !samps = nsamps `div` 4
+   !org = Vec 50 52 295.6
+   !dir = norm $ Vec 0 (-0.042612) (-1)
+   !cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
+   !cy = norm (cx `cross` dir) `mulvs` 0.5135
...
-   r1 <- (2*) `fmap` erand48 xi
-   let dx = if r1<1 then sqrt r1-1 else 1-sqrt(2-r1)
-   r2 <- (2*) `fmap` erand48 xi
-   let dy = if r2<1 then sqrt r2-1 else 1-sqrt(2-r2)
-   d = ...
-   rad <- radiance (Ray (org`addv`(d`mulvs`140)) (norm d)) 0 xi
+   !r1 <- (2*) `fmap` erand48 xi
+   let !dx = if r1<1 then sqrt r1-1 else 1-sqrt(2-r1)
+   !r2 <- (2*) `fmap` erand48 xi
+   let !dy = if r2<1 then sqrt r2-1 else 1-sqrt(2-r2)
+   !d = ...
...
+       pure $! r + rad .* recip (fromIntegral samps)
+       pure $! ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
```

Reduce to only useful strictnesses in smallpt(1.17x)

```
- let !samps = nsamps `div` 4
- !org = Vec 50 52 295.6
- !dir = norm $ Vec 0 (-0.042612) (-1)
- !cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
- !cy = norm (cx `cross` dir) .* 0.5135
+ let samps = nsamps `div` 4
+ org = Vec 50 52 295.6
+ dir = norm $ Vec 0 (-0.042612) (-1)
+ cx = Vec (fromIntegral w * 0.5135 / fromIntegral h) 0 0
+ cy = norm (cx `cross` dir) .* 0.5135
...
- !r1 <- (2*) <$> erand48
+ r1 <- (2*) <$> erand48
...
- !r2 <- (2*) <$> erand48
+ r2 <- (2*) <$> erand48
...
- !rad <- radiance (Ray (org+d.*140) (norm d)) 0
+ rad <- radiance (Ray (org+d.*140) (norm d)) 0
...
- pure $! r + rad .* recip (fromIntegral samps)
- pure $! ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25
+ pure (r + rad .* recip (fromIntegral samps))
+ pure (ci + Vec (clamp rr) (clamp rg) (clamp rb) .* 0.25)
```


Use strictness strategically in entire project

```
...  
- if det<0 then Nothing else f (b-sdet) (b+sdet)  
- where op = p - o  
-       eps = 1e-4  
-       b = dot op d  
-       det = b*b - dot op op + r*r  
-       sdet = sqrt det  
-       f a s = if a>eps then Just a else if s>eps then Just s else Nothing  
+ if det<0  
+ then Nothing  
+ else  
+   let !eps = 1e-4  
+       !sdet = sqrt det  
+       !a = b-sdet  
+       !s = b+sdet  
+   in if a>eps then Just a else if s>eps then Just s else Nothing  
...
```

Remove Maybe from intersect(s) (1.32x)

```
-intersect :: Ray -> Sphere -> Maybe Double
+intersect :: Ray -> Sphere -> Double
intersect (Ray o d) (Sphere r p _e _c _refl) =
-   if det<0 then Nothing else f (b-sdet) (b+sdet)
+   if det<0 then (1/0.0) else f (b-sdet) (b+sdet)
  where op = p `subv` o
        eps = 1e-4
        b = op `dot` d
        det = b*b - (op `dot` op) + r*r
        sdet = sqrt det
-   f a s = if a>eps then Just a else if s>eps then Just s else Nothing
+   f a s = if a>eps then a else if s>eps then s else (1/0.0)

-intersects :: Ray -> (Maybe Double, Sphere)
+intersects :: Ray -> (Double, Sphere)
intersects ray = (k, s)
-   where (k,s) = foldl' f (Nothing,undefined) spheres
-         f (k',sp) s' = case (k',intersect ray s') of
-             (Nothing,Just x) -> (Just x,s')
-             (Just y,Just x) | x < y -> (Just x,s')
-             _ -> (k',sp)
+   where (k,s) = foldl' f (1/0.0,undefined) spheres
+         f (k', sp) s' = let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)

radiance :: Ray -> Int -> STRefU s Word64 -> ST s Vec
radiance ray@(Ray o d) depth xi = case intersects ray of
-   (Nothing,_) -> return zerov
-   (Just t,Sphere _r p e c refl) -> do
+   (t,_) | t == (1/0.0) -> return zerov
+   (t,Sphere _r p e c refl) -> do
```

Hand unroll the fold in intersects (1.35x)

```
intersects :: Ray -> (Double, Sphere)
-intersects ray = (k, s)
-  where (k,s) = foldl' f (1/0.0,undefined) spheres
+intersects ray =
+  f (... (f (f (intersect ray sphLeft, sphLeft) sphRight) ...))
+  where
+    f (k', sp) s' = let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)

-spheres :: [Sphere]
-spheres = let s = Sphere ; z = zerov ; (.* ) = mulvs ; v = Vec in
-  [ s 1e5 (v (1e5+1) 40.8 81.6)    z (v 0.75 0.25 0.25) DIFF --Left
-    , s 1e5 (v (-1e5+99) 40.8 81.6) z (v 0.25 0.25 0.75) DIFF --Right
-  ...

+sphLeft, sphRight, ... :: Sphere
+sphLeft  = Sphere 1e5  (Vec (1e5+1) 40.8 81.6)    zerov (Vec 0.75 0.25 0.25) DIFF
+sphRight = Sphere 1e5  (Vec (-1e5+99) 40.8 81.6)   zerov (Vec 0.25 0.25 0.75) DIFF
+...
```

Custom datatype for intersects parameter passing

```
-intersects :: Ray -> (Double, Sphere)
+data T = T !Double !Sphere
+
+intersects :: Ray -> T
intersects ray =
-   f ( ... f (intersect ray sphLeft, sphLeft) sphRight) ... sphLite
+   f ( ... f (T (intersect ray sphLeft) sphLeft) sphRight) ... sphLite
  where
-   f (k', sp) s' =
-       let !x = intersect ray s' in if x < k' then (x, s') else (k', sp)
+   f !(T k' sp) !s' =
+       let !x = intersect ray s' in if x < k' then T x s' else T k' sp

radiance :: Ray -> Int -> Erand48 Vec
radiance ray@(Ray o d) depth = case intersects ray of
-   (!t,_) | t == 1/0.0 -> return 0
-   (!t,!Sphere _r p e c refl) -> do
+   (T t _) | t == 1/0.0 -> return 0
+   (T t (Sphere _r p e c refl)) -> do
    let !x = o + d .* t
        !n = norm $ x - p
        !nl = if dot n d < 0 then n else negate n
```

Optimize file writing

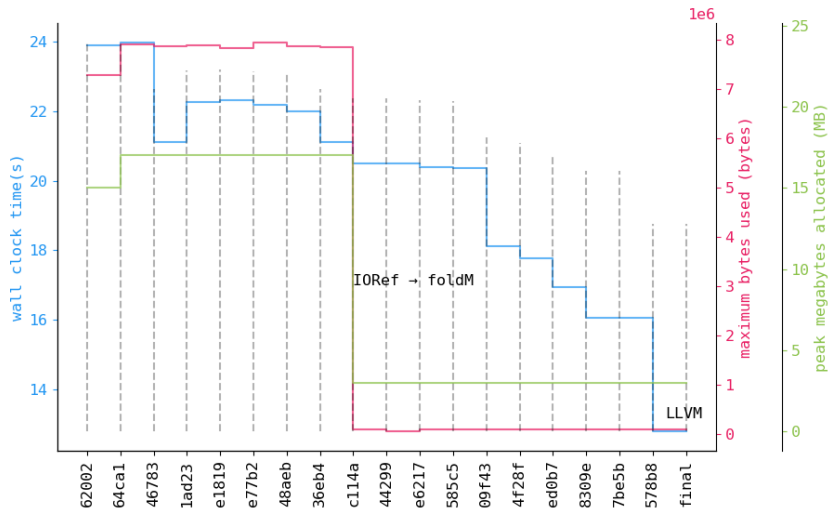
```
build-depends:
    base >= 4.12 && < 4.15
+   , bytestring ^>= 0.11

-toInt :: Double -> Int
-toInt x = floor $ clamp x ** recip 2.2 * 255 + 0.5
+toInt :: Double -> BB.Builder
+toInt x = BB.intDec (floor (clamp x ** recip 2.2 * 255 + 0.5)) <> BB.char8 ' '
...
withFile "image.ppm" WriteMode $ \hdl -> do
-   hPrintf hdl "P3\n%d %d\n%d\n" w h (255::Int)
-   for_ img \(Vec r g b) -> do
-       hPrintf hdl "%d %d %d " (toInt r) (toInt g) (toInt b)
+   BB.hPutBuilder hdl $
+       BB.string8 "P3\n" <>
+       BB.intDec w <> BB.char8 ' ' <> BB.intDec h <> BB.char8 '\n' <>
+       BB.intDec 255 <> BB.char8 '\n' <>
+       (mconcat $ fmap \(Vec r g b) -> toInt r <> toInt g <> toInt b) img)
```

Use LLVM backend (1.87×)

```
+package smallpt-opt + ghc-options: -fllvm
```

The view from the mountaintop



Takeaways

- ▶ The unrolling in 'intersects' is ugly.
- ▶ (We feel) the maintainability of this code hasn't been significantly harmed.
- ▶ We're faster than clang++ and within 6% of g++
- ▶ Haven't exhausted the optimization opportunities.
- ▶ GHC could learn to do several of these optimizations for us.
- ▶ Others are just good Haskell style.
- ▶ Clean Haskell is often performant Haskell.