

Optimizing smallpt

Daven Scies, Siddharth Bhat

Haskell Exchange

November 4th, 2020

What is smallpt anyway?

```

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
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 ...
};
inline bool intersect(const Ray &r, double &t, int &id)

```

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);
}

```

Haskell: the first stab

How do we communicate the code? x(

```

radiance :: Ray -> CInt -> Ptr CUSHort -> IO Vec
radiance ray@(Ray o d) depth xi =
  case intersects ray of
    (Nothing,_) -> return zeroV
    (Just (D# t),Sphere r p e c refl) -> do
      let x = addv o (mulvs d t)
          n = norm $ x `subv` p
          n1 = if isTrue# ((dot n d) <## 0.0##) then n else mulvs n (-1.0##)
          pr = maxv c
          depth' = depth + 1
          continue f = case refl of
            DIFF -> do
              (CDouble (D# r)) <- erand48 xi
              let r1 = (2.0## *** 3.141592653589793238##) *** r
                  (CDouble (D# r2)) <- erand48 xi
                  let r2s = sqrtDouble# r2
                      w@(Vec wx _ _) = n1
                      u = norm (cross (if isTrue# (fabsDouble# wx >## 0.1##) then (Vec 0.0## 1.0## 0.0##) else (Vec 1.0## 0.0## 0.0##))
                          v = w `cross` u
                          d' = norm $ (u`mulvs`(cosDouble# r1*##r2s)) `addv` (v`mulvs`(sinDouble# r1*##r2s)) `addv` (w`mulvs`sqrtDouble# r2s)
              rad <- radiance (Ray x d') depth' xi
              return $ e `addv` (f `mulv` rad)
            SPEC -> do
              let d' = d `subv` (n `mulvs` (2.0## *** (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` (2.0## *** n`dot`d))) -- Ideal dielectric REFRACTION
                  into = n`dot`n1 >## 0.0## -- Ray from outside going in?
                  nc = 1.0##
                  nt = 1.5##
                  nnt = if isTrue# into then nc/##nt else nt/##nc

```

Optimisation 1: manual unrolling + unboxing

Optimisation 2: Optimizing Ray

Optimisation 3: Newtyping Ref1

Optimisation 4: Unbox tuple of intersects

Optimisation 3: Only expose main

Optimisation 3: Strictify intersect

Optimisation ...: Enable LLVM

```
diff --git a/smallpt-hs.cabal b/smallpt-hs.cabal
index 83ec118..64d2788 100644
--- a/smallpt-hs.cabal
+++ b/smallpt-hs.cabal
@@ -26,4 +26,5 @@ executable smallpt-hs
                                vector

    -- no -Wall as type signature are purposely missing
+  ghc-options:                 -O2 -rtsopts -fllvm
-  ghc-options:                 -O2
```

Takeaways

Takeaways

- Haskell can be fast

Takeaways

- Haskell can be fast
- ... with a lot of work!

Takeaways

- Haskell can be fast
- ... with a lot of work!
- Accumulate optimizations to accrue performance wins.

Takeaways

- Haskell can be fast
- ... with a lot of work!
- Accumulate optimizations to accrue performance wins.
- [Raw Google Sheet](#)
- github.com/bollu/smallpths