Howard

Justin Peng and Akash Nayar

12/20/2023

1 Introduction

Howard is a ray tracer written in Haskell based on the Ray Tracing in One Weekend book by Peter Shirley, Trevor David Black, and Steve Hollasch. Howard is capable of rendering 3-dimensional scenes of spheres composed of metal, diffuse, or dielectric materials.

1.1 Ray Tracing

With the advancement of graphical processing hardware, light rendering algorithms have seen increasing popularity in recent years. One such popular method that has gained significant popularity in the past decade is ray tracing. Ray tracing simulates individual "rays" of light, tracking their movement through space and how they interact with objects, being reflected, scattered, or refracted. The propagation of rays is represented as a function of time. Oftentimes, millions of rays are propagated throughout the scene and their interaction with the environment is calculated. In ray tracing, a camera and view port facing the scene are established. Then the direction of each ray from the center of the camera based on the view port is calculated and is sent through every pixel of the generated image.

1.2 Potential for Parallelization

Ray tracing presents an interesting candidate for parallelization. The nature of individually simulated light rays and time-stepping methods allow us to subdivide the work of the simulation between threads. Because the work performed to calculate the propagation and color of each ray is relatively similar, it is possible to simultaneously calculate the propagation of rays. Implementations that subdivide the generation of the image, the propagation of rays, or intersection calculations with objects in the scene provide viable candidates for parallelization.

2 Implementation

2.1 Rays

In Howard, rays are considered a function of time given an origin and direction. We can think of a ray as the equation

$$\vec{P}(t) = \vec{A} + t\vec{b}$$

To represent the equation, rays are represented in Howard by a data type that contains the origin and direction of the ray. Prior to the rendering of the scene, a view port is established in front of the objects that are to be rendered. A ray is created originating from the center of the camera toward each pixel in the scene. We can compute the direction of the ray given the width and height of the view port. Afterwards, we can check whether or not each ray intersects with an object in the scene at a given time. To simulate the propagation of the rays, we iterate through each pixel of the generated image, calculate the direction of the ray, and check for collisions with any of the objects in the scene.

```
data Ray = Ray
  {
    origin :: Vec3,
    direction :: Vec3
  } deriving (Show)
```

2.2 Collisions

Ray tracing seeks to model the interaction between light rays and objects in the scene. To do this, the point[s] at which the ray intersects with the object must be calculated. Additionally, to compute other interactions, such as the angle of reflection or refraction of light, the normal of the object at the point of intersection must be computed as well.

2.2.1 Sphere Intersection

Howard implements intersections with spheres. To compute whether the light ray intersects with the sphere and whether the light has one or two intersections, Howard uses the equation for a sphere in $\mathbb{R}^3$:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

We know that a ray has intersected the circle if the current position of the ray satisfies the above equation. Knowing this, we can represent the above equation in terms of the current position:

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \Rightarrow \left(\vec{P} - \vec{C}\right) \cdot \left(\vec{P} - \vec{C}\right) = r^2$$

where P is the current position of the ray, and C is the center of the circle. Therefore, substituting in $P = \vec{A} + t\vec{b}$, we get:

$$t^2\vec{b} \cdot \vec{b} \cdot \left(\vec{A} - \vec{C}\right) + \left(\vec{A} - \vec{C}\right) \cdot \left(\vec{A} - \vec{C}\right) - r^2 = 0$$

Thus, because we know the initial position and direction of the rays, we can solve for t. If there is no real solution, there is no intersection, and otherwise, the number of real solutions is the number of intersections.

2.3 Anti-aliasing

A common issue when representing high-dimensional objects in lower-dimensional images is the presence of "aliasing", or a jagged appearance along the edges of objects. To create the appearance of a "smoother" image, when determining the color of a pixel, the color of surrounding pixels is randomly sampled, and their color values are averaged. Therefore, if a pixel is on the perceived edge between an object and the background, the resulting color will be approximately the average of the object and the background.

2.4 Materials

2.4.1 Diffuse

Diffuse materials in Howard follow true Lambertian reflection and are defined by an albedo. When a ray collides with a diffuse material, the resulting scattered ray $\vec{s}$ is given by $\vec{s} = \hat{n} + \hat{r}$, where $\hat{n}$ is the unit normal vector of the surface and $\hat{r}$ is a random unit vector. In the degenerate case where $\|\vec{s}\| < \epsilon$ ($\epsilon = 10^{-6}$), we just return $\vec{s} = \hat{n}$. The scattered ray is tinted with the material's albedo.

```haskell
data Lambertian = Lambertian Vec3
instance Material Lambertian where
  scatter _ (HitRecord p' n' _ _ _) g (Lambertian albedo) =
    (Just (Ray p' scatter_direction, albedo), g1)
      where
        (rand, g1) = randomUnitVector g
        new_scatter = n' `addVec3` rand
        scatter_direction = if (nearZero new_scatter) then n' else (new_scatter)
```

2.4.2 Metal

Metals in Howard are defined with an albedo and a "fuzzy" parameter ranging from 0 to 1. The fuzziness of a metallic object represents how strongly a scattered ray will deviate from pure reflection. Higher fuzziness will create more matte-like metals, whereas a fuzziness of 0 will imitate a mirror. The equation for a scattered ray $\vec{s}$ is given by $\vec{s} = \hat{g} + f * \hat{r}$, where f is the fuzziness, $\hat{r}$ is a random unit vector, and $\hat{g}$ is the reflection of the incoming ray ($\hat{b}$) over the surface normal, given by $\hat{g} = \hat{b} - (2\hat{b} \cdot \hat{n})\hat{n}$.

```haskell
data Metal = Metal Vec3 Double
instance Material Metal where
  scatter ray (HitRecord p' n' _ _ _) g (Metal albedo fuzz) =
    if ((direction scattered) `dot` n' > 0) then (Just (scattered, albedo), g1) else (Nothing, g1)
      where
        reflected = reflect (unitVector (direction ray)) n'
        (rand, g1) = randomUnitVector g
        scattered = (Ray p' (reflected `addVec3` (rand `multiplyVec3` fuzz)))

reflect :: Vec3 -> Vec3 -> Vec3
reflect v n = v `minusVec3` (n `multiplyVec3` (2 * (v `dot` n)))
```

2.4.3 Dielectric

Dielectrics in Howard are implemented using Snell's Law and Schlick Approximation, and have the index of refraction as their sole parameter. We begin by defining constants c and s as follows:

$$
\begin{aligned}
c &= \min(-\hat{b} \cdot \hat{n},\, 1) \\
s &= \sqrt{1 - c^2}
\end{aligned}
$$

In the above equations, $\hat{b}$ is the unit direction of the incoming ray, and $\hat{n}$ is the unit normal vector of the dielectric surface it is colliding with. If we are colliding with the outer surface of an object, the refraction ratio f is the inverse of the index of refraction of that object. Otherwise, it is simply the index of refraction of that object. If $f * s > 1$, we cannot refract, and will instead simply reflect the ray across the normal. In addition, there is some probability that a given ray will get reflected off a dielectric surface instead of refracted. We can compute this probability using Schlick's Approximation. Otherwise, we obtain the refracted direction $\vec{p}$, given by the equation $\vec{p} = \vec{p}_{\perp} + \vec{p}_{\|}$. $\vec{p}_{\perp}$ and $\vec{p}_{\|}$ are given by the following equations:

$$
\begin{aligned}
\vec{p}_{\perp} &= \frac{\eta}{\eta'}(\hat{b} + (\hat{b} \cdot \hat{n})\hat{n}) \\
\vec{p}_{\|} &= -\sqrt{1 - \|\vec{p}_{\perp}\|^2}\hat{n}
\end{aligned}
$$

In the above equations, η and η' are the indexes of refraction of the previous and current materials, respectively. $\hat{b}$ is the direction of the incoming ray and $\hat{n}$ is the normal vector of the surface at the intersection point.

```haskell

data Dielectric = Dielectric Double
instance Material Dielectric where
    scatter ray (HitRecord p' n' _ _ f') g (Dielectric ir) =
        (Just (scattered, color), g1)
            where
                color = Vec3 1.0 1.0 1.0
                refractionRatio = if f' then 1.0 / ir else ir
                unitDirection = unitVector (direction ray)
                cosTheta = min (negateVec3 unitDirection `dot` n
   ') 1.0
                sinTheta = sqrt (1.0 - cosTheta * cosTheta)
                cannotRefract = refractionRatio * sinTheta > 1.0
                (rd, g1) = randomDouble g
                scattered = if cannotRefract || reflectance
   cosTheta refractionRatio > rd then Ray p' (reflect
   unitDirection n') else Ray p' (refract unitDirection n'
   refractionRatio)
```

```


reflectance :: Double -> Double -> Double
reflectance cosine refIdx = ret
    where
        r0 = (1 - refIdx) / (1 + refIdx)
        r0' = r0 * r0
        ret = r0' * (1 + r0')*(1 - cosine)**5

refract :: Vec3 -> Vec3 -> Double -> Vec3
refract uv n refrac = rOutPerp `addVec3` rOutParallel
  where
    cosTheta = min (negateVec3 uv `dot` n) 1.0
    rOutPerp = (uv `addVec3` (n `multiplyVec3` cosTheta)) `multiplyVec3` refrac
    rOutParallel = n `multiplyVec3` ((-1) * sqrt (abs (1.0 - (lengthSquaredVec3 rOutPerp))))
```

3 Parallelism

Due to the highly individualized and relatively balanced nature of the workload of Howard, we can easily parallelize the rendering of the image. Specifically, for each pixel that is rendered, Howard checks for a collision with every object in the scene, determining the closest collision before calculating the resulting reflection/refraction and color of the ray. Thus, we can see that we can parallelize these operations by simultaneously performing these calculations across multiple threads.

3.1 Parallel Implementation

The parallel processing of rays is facilitated via the **Control.Parallel** library. In its parallel implementation, Howard splits the rendering of the image into rows, mapping a function that handles the ray propagation for every pixel in a given row index over a list of row indices. We use the **parMap** function from **Control.Parallel.Strategies** to spark a parallel evaluation of each row corresponding with the row index. Furthermore, we can use **rdeepseq** to force the full evaluation/rendering of each row before the final image starts generating. After each row has been rendered, the resulting colors from each rendered pixel are processed in order.

```
renderParallel :: Hittable a => Camera -> a -> IO()
renderParallel cam world = do
  putStrLn $ "P3\n" ++ show (imageWidth cam) ++ " " ++ show (imageHeight cam) ++ "\n255"
  let rows = [0..imageHeight cam - 1]
  let processedRows = parMap rdeepseq (processRow cam world) rows
  mapM_ putStrLn processedRows

processRow :: Hittable a => Camera -> a -> Int -> String
processRow cam world j = unlines $ map (processPixel cam world j) [0..imageWidth cam - 1]

processPixel :: Hittable a => Camera -> a -> Int -> Int -> String
processPixel cam world j i =
    let (pixelColor, _) = updateColor (samplesPerPixel cam) (Vec3 0 0 0) i j cam world (mkStdGen (i * (imageHeight cam - 1) + j))
    in writeColorStr pixelColor (samplesPerPixel cam)
```

4 Performance

To benchmark the performance of the parallelized version of Howard vs. the sequential version, we measured the runtime of the two implementations across five different scenes. Each scene was rendered with a width of 720 pixels, an aspect ratio of 16:9 for a height of 405 pixels, and a sampling size of 100. Each benchmarked time is an average of five runs with the parallel and sequential implementations, performed on a 10-core Apple M2 Pro processor.

4.1 Benchmark Scenes

We created five scenes with different numbers of spheres of varying materials to test Howard's performance. Each scene consists of a large ground sphere and one to three additional spheres.

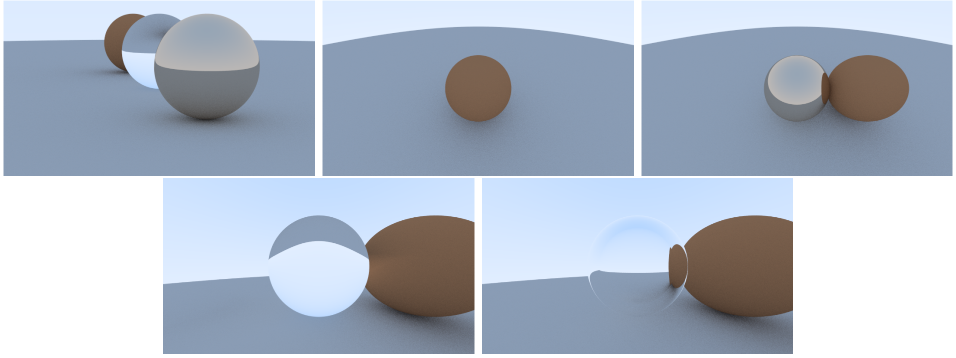

Figure 1: Default, Diffuse, Metal, Dielectric, and Hollow Glass Sphere scenes (from left to right, top to bottom)

4.1.1 Load Balancing

To examine load balancing, the event logs of the sequential and parallel implementations were run using an Intel™ i7-10750H and examined on ThreadScope. The specific scene examined was the Diffuse scene. The ThreadScope results of the single-threaded implementation show very consistent performance, with very little garbage collection.

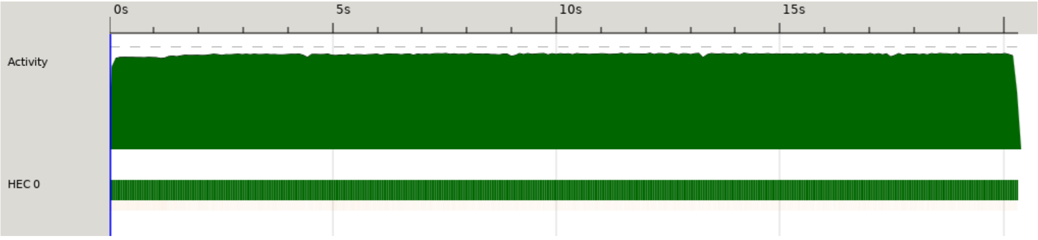

Figure 2: ThreadScope: Single Thread

As we increase the number of cores to two, we notice that we are able to effectively balance the load between threads. Both cores are active throughout the entire runtime, and although garbage collection has slightly increased, it has not significantly impacted the performance.

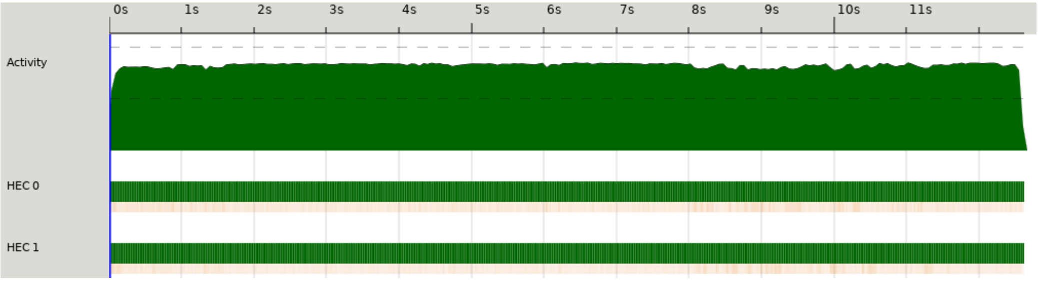

Figure 3: ThreadScope: Two Threads

However, if we increase the number of cores significantly, the amount of garbage collection per thread noticeably increases. Additionally, while each core remains active for the duration of the runtime, there is a small moment near the end of the execution where a few threads wait. This suggests that there may be some inequality in the amount of work each thread does. We can see that compared to using two cores, the speed has not significantly improved, suggesting that the garbage collection has tangibly impacted performance.

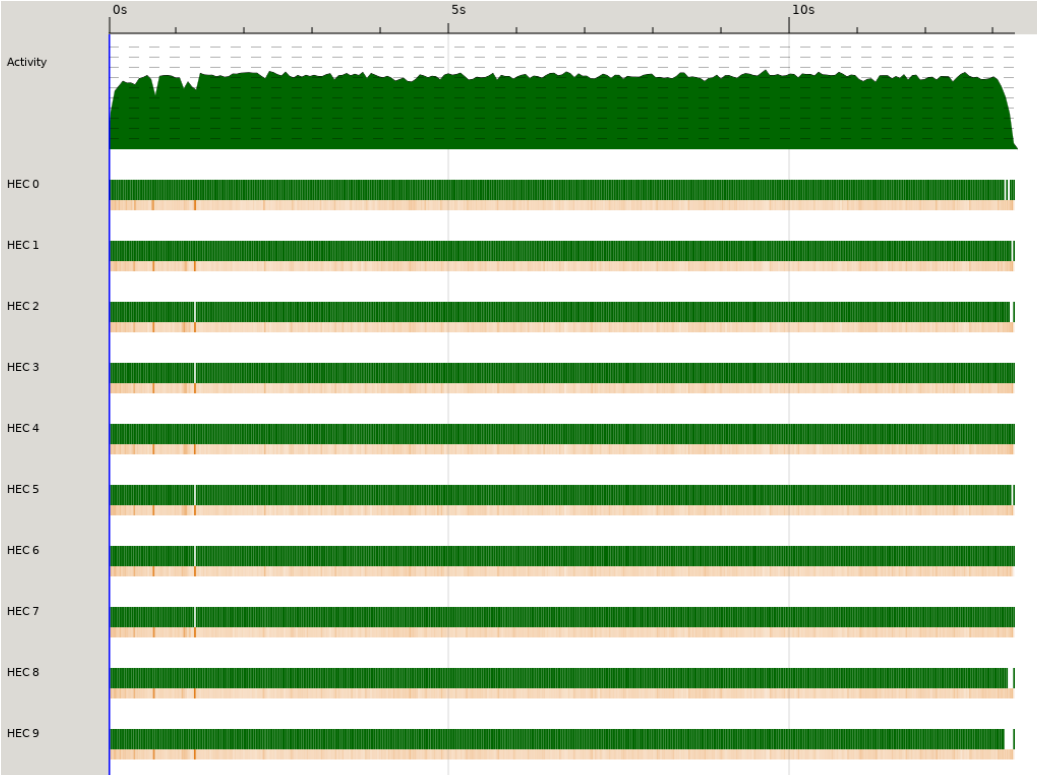

Figure 4: ThreadScope: Ten Threads

4.2 Performance Results

Overall, parallelizing Howard greatly increased its performance. Performance plateaued after 6 cores, for an average speedup of 487%.

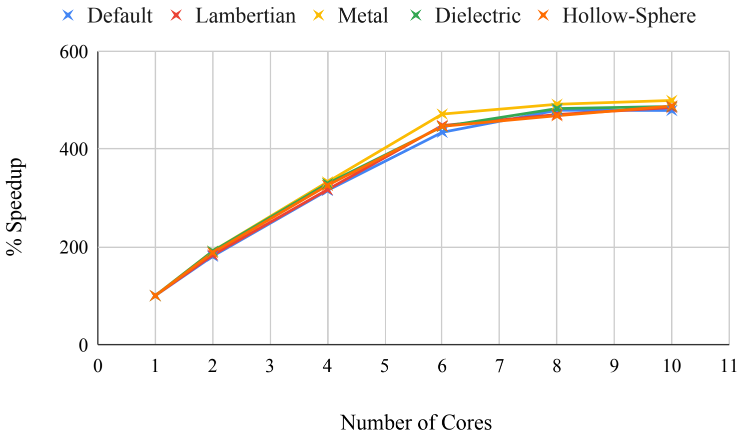

Figure 5: % Speedup vs. Number of Cores

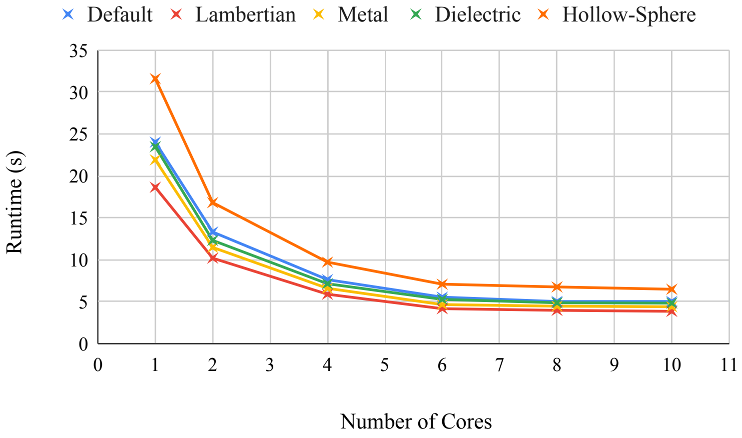

Figure 6: Runtime vs. Number of Cores

5 Code

All of Howard's code can be found on Github. The code we wrote is located in the /app and /src subdirectories. A README explaining how to run and test Howard is available in the Github repository.

app/Main.hs:

```haskell
module Main (main) where

import Vec3
import Sphere
import Hittable
import Camera
import Utilities
import System.Random
import System.Environment
import Data.Maybe (catMaybes)

randomLambert :: Vec3 -> StdGen -> (Sphere, StdGen)
randomLambert center g = (Sphere center 0.2 mat, g1)
  where
    (v, g1) = randomVec3 g
    mat = Lambertian v

randomMetal :: Vec3 -> StdGen -> (Sphere, StdGen)
randomMetal center g = (Sphere center 0.2 mat, g2)
  where
    (v, g1) = randomVec3 g
    (f, g2) = randomDouble g1
    mat = Metal v f

dielectric :: Vec3 -> Sphere
dielectric center = Sphere center 0.2 (Dielectric 1.5)

randomBall :: Int -> Int -> StdGen -> Maybe Sphere
randomBall a b g =
  if lengthVec3 (center `minusVec3` (Vec3 4 0.2 0)) < 0.9
      then Nothing
      else Just sphere
        where
          (chooseMat, g1) = randomDouble g
          (x', g2) = randomDouble g1
          (y', g3) = randomDouble g2
```

```haskell
          center = Vec3 (fromIntegral a + 0.9 * x') 0.2 (fromIntegral b + 0.9 * y')
          (l, g4) = randomLambert center g3
          (m, _) = randomMetal center g4
          sphere
            | chooseMat < 0.8 = l
            | chooseMat < 0.95 = m
            | otherwise = dielectric center

main :: IO ()
main = do
  args <- getArgs
  let
    notParallel = if not (null args) then (if head args == "single" then True else False) else False
    remainingArgs = if not (null args) then (if head args == "single" then tail args else args) else args
    scene = if not (null remainingArgs) then head remainingArgs else ""
    material_ground = Lambertian (Vec3 0.5 0.5 0.5)
    objects = (catMaybes [randomBall a b (mkStdGen (21 * a + b)) | a <- [-11,-10..11], b <- [-11,-10..11]])

    material1 = Dielectric 1.5
    material2 = Lambertian (Vec3 0.4 0.2 0.1)
    material3 = Metal (Vec3 0.7 0.6 0.5) 0.0
    width = 720
    samples = 100
  case scene of
    "final" -> do
      let
        groundSphere = Sphere (Vec3 0 (-1000) 0) 1000 material_ground
        sphere3 = Sphere (Vec3 0 1 0) 1.0 material1
        sphere2 = Sphere (Vec3 (-4) 1 0) 1.0 material2
        sphere1 = Sphere (Vec3 4 1 0) 1.0 material3

        world = HittableList ([sphere1, sphere2, sphere3] ++ objects ++ [groundSphere])

        vFov = 20
        lookFrom = Vec3 13 2 3
        lookAt = Vec3 0 0 0
        vUp = Vec3 0 1 0
```

```
        cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
      case notParallel of
        True -> render cam world
        False -> renderParallel cam world
    "lambertian" -> do
      let

        groundSphere = Sphere (Vec3 0 (-100.5) 0) 100 material_ground
        sphere1 = Sphere (Vec3 0 0 (-1)) 0.5 material2

        world = HittableList [sphere1, groundSphere]

        vFov = 90
        lookFrom = Vec3 0 1 0
        lookAt = Vec3 0 0 (-1)
        vUp = Vec3 0 1 0
        cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
      case notParallel of
        True -> render cam world
        False -> renderParallel cam world
    "metal" -> do
      let
        groundSphere = Sphere (Vec3 0 (-100.5) 0) 100 material_ground
        sphere1 = Sphere (Vec3 0 0 (-1)) 0.5 material3
        sphere2 =  Sphere (Vec3 1 0 (-1)) 0.5 material2
        world = HittableList [sphere1, sphere2, groundSphere]

        vFov = 90
        lookFrom = Vec3 0 1 0
        lookAt = Vec3 0 0 (-1)
        vUp = Vec3 0 1 0
        cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
      case notParallel of
        True -> render cam world
        False -> renderParallel cam world
    "dielectric" -> do
      let
        groundSphere = Sphere (Vec3 0 (-100.5) 0) 100 material_ground
        sphere1 = Sphere (Vec3 0 0 (-1)) 0.5 material1
```

```
      sphere2 =  Sphere (Vec3 1 0 (-1)) 0.5 material2
      world = HittableList [sphere1, sphere2, groundSphere]

      vFov = 90
      lookFrom = Vec3 0 0 0
      lookAt = Vec3 0 0 (-1)
      vUp = Vec3 0 1 0
      cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
    case notParallel of
      True -> render cam world
      False -> renderParallel cam world
  "hollow-sphere" -> do
    let
      groundSphere = Sphere (Vec3 0 (-100.5) 0) 100 material_ground
      sphere1 = Sphere (Vec3 0 0 (-1)) 0.5 material1
      sphere1Inner = Sphere (Vec3 0 0 (-1)) (-0.4) material1
      sphere2 =  Sphere (Vec3 1.01 0 (-1)) 0.5 material2
      world = HittableList [sphere1, sphere1Inner, sphere2, groundSphere]

      vFov = 90
      lookFrom = Vec3 0 0 0
      lookAt = Vec3 0 0 (-1)
      vUp = Vec3 0 1 0
      cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
    case notParallel of
      True -> render cam world
      False -> renderParallel cam world
  _ -> do
    let
      groundSphere = Sphere (Vec3 0 (-1000) 0) 1000 material_ground
      sphere3 = Sphere (Vec3 0 1 0) 1.0 material1
      sphere2 = Sphere (Vec3 (-4) 1 0) 1.0 material2
      sphere1 = Sphere (Vec3 4 1 0) 1.0 material3

      world = HittableList [sphere1, sphere2, sphere3, groundSphere]

      vFov = 20
      lookFrom = Vec3 13 2 3
      lookAt = Vec3 0 0 0
```

```
        vUp = Vec3 0 1 0

        cam = initialize (16.0/9.0) width samples vFov lookFrom lookAt vUp
      case notParallel of
        True -> render cam world
        False -> renderParallel cam world
```

src/Camera.hs:

```
module Camera(
 Camera (Camera),
 initialize,
 rayColor,
 renderParallel,
 render
) where

import Vec3
import Ray
import Hittable
import Interval
import Color
import Utilities
import System.Random (mkStdGen, StdGen)
import Control.Parallel.Strategies
data Camera = Camera
 {
  aspectRatio :: Double,
  imageWidth :: Int,
  imageHeight:: Int,
  samplesPerPixel :: Int,
  center :: Vec3,
  pixel100Loc :: Vec3,
  pixelDeltaU :: Vec3,
  pixelDeltaV :: Vec3,
  maxDepth :: Int
 } deriving Show


initialize :: Double -> Int -> Int -> Double -> Vec3 -> Vec3 -> Vec3 -> Camera
initialize aspect width samples vFov lookFrom lookAt vUp = Camera aspect width height samples cent pixel100 deltaU deltaV 50
       where
```

```haskell
        height = max 1 (floor $ fromIntegral width / aspect)

        cent = lookFrom
        focalLength = lengthVec3 (lookFrom `minusVec3` lookAt)

        w = unitVector (lookFrom `minusVec3` lookAt)
        u = unitVector (cross vUp w)
        v = cross w u

        theta = degreesToRadians vFov
        h = tan (theta / 2.0)
        viewPortHeight = 2.0 * h * focalLength
        viewPortWidth = viewPortHeight * (fromIntegral width / fromIntegral height)

        viewPortU = u `multiplyVec3` viewPortWidth
        viewPortV = (negateVec3 v) `multiplyVec3` viewPortHeight

        deltaU = viewPortU `divideVec3` fromIntegral width
        deltaV = viewPortV `divideVec3` fromIntegral height

        viewPortUpperLeft = cent `minusVec3` (w `multiplyVec3` focalLength) `minusVec3` (viewPortU `divideVec3` 2) `minusVec3` (viewPortV `divideVec3` 2)
        pixel100 = viewPortUpperLeft `addVec3` ((deltaU `addVec3` deltaV) `multiplyVec3` (0.5 :: Double))

rayColor :: Hittable a => Ray -> a -> Int -> StdGen -> (Vec3, StdGen)
rayColor _ _ 0 g = (Vec3 0 0 0, g)
rayColor (Ray org dir) world i g = ret
    where
     isHit = hit (Ray org dir) (Interval 0.001 9999999999999) Nothing world
     unit_direction = unitVector dir
     a = (y unit_direction + 1.0) * 0.5
     ret = case isHit of
      Nothing -> ((Vec3 1.0 1.0 1.0 `multiplyVec3` (1.0 - a)) `addVec3` (Vec3 0.5 0.7 1.0 `multiplyVec3` a), g)
      Just (HitRecord p2 n2 m t2 ff) ->
       case scatter (Ray org dir) (HitRecord p2 n2 m t2 ff) g m of
        (Nothing, g1) -> (Vec3 0 0 0, g1)
        (Just (scattered, attenuation), g1) -> (attenuation `multiplyVec3` v, g2)
```

```
         where
          (v, g2) = (rayColor scattered world (i - 1) g1)

renderParallel :: Hittable a => Camera -> a -> IO()
renderParallel cam world = do
 putStrLn $ "P3\n" ++ show (imageWidth cam) ++ " " ++ show (
   imageHeight cam) ++ "\n255"
 let rows = [0..imageHeight cam - 1]
 let processedRows = parMap rdeepseq (processRow cam world) rows
 mapM_ putStrLn processedRows

render :: Hittable a => Camera -> a -> IO()
render cam world = do
    putStrLn $ "P3\n" ++ show (imageWidth cam) ++ " " ++ show (
   imageHeight cam) ++ "\n255"
    mapM_ (\j -> mapM_ (\i -> do
      let (pixelColor, _) = updateColor (samplesPerPixel cam) (
   Vec3 0 0 0) i j cam world (mkStdGen (i * (imageHeight cam -
   1) + j))
      writeColor pixelColor (samplesPerPixel cam)
     ) [0..imageWidth cam -1]) [0..imageHeight cam-1]

processRow :: Hittable a => Camera -> a -> Int -> String
processRow cam world j = unlines $ map (processPixel cam world j
   ) [0..imageWidth cam - 1]

processPixel :: Hittable a => Camera -> a -> Int -> Int ->
   String
processPixel cam world j i =
 let (pixelColor, _) = updateColor (samplesPerPixel cam) (Vec3 0
    0 0) i j cam world (mkStdGen (i * (imageHeight cam - 1) + j)
   )
 in writeColorStr pixelColor (samplesPerPixel cam)

updateColor :: Hittable a => Int -> Vec3 -> Int -> Int -> Camera
    -> a -> StdGen -> (Vec3, StdGen)
updateColor 0 x1 _ _ _ _ g = (x1, g)
updateColor samples cur i j cam world g = updateColor (samples -
    1) next i j cam world g2
           where
            (r, g1) = getRay cam i j g
            (rc, g2) = rayColor r world (maxDepth cam) g1
            next = cur `addVec3` rc

getRay :: Camera -> Int -> Int -> StdGen -> (Ray, StdGen)
```

```
getRay cam i j g = (Ray org dir, g1)
    where
     pixelCenter = pixel100Loc cam `addVec3` (pixelDeltaU cam `multiplyVec3` (fromIntegral i :: Double)) `addVec3` (pixelDeltaV cam `multiplyVec3` (fromIntegral j :: Double))
     (pss, g1) = pixelSampleSquare cam g
     pixelSample = pixelCenter `addVec3` pss

     org = center cam
     dir = pixelSample `minusVec3` org

pixelSampleSquare :: Camera -> StdGen -> (Vec3, StdGen)
pixelSampleSquare cam g = res
     where
      px = -0.5 + d
      py = -0.5 + d1
      (d, g1) = randomDouble g
      (d1,g2) = randomDouble g1
      res = ((pixelDeltaU cam `multiplyVec3` px) `addVec3` (pixelDeltaV cam `multiplyVec3` py), g2)
```

src/Color.hs:

```
module Color(
  writeColorStr,
  writeColor
) where

import Vec3
import Interval

linearToGamma :: Double -> Double
linearToGamma linearCompart = sqrt linearCompart

writeColor :: Vec3 -> Int -> IO()
writeColor (Vec3 r g b) samples = do

        let scale = 1.0 / fromIntegral samples
          rScaled = r * scale
          gScaled = g * scale
          bScaled = b * scale

          rGamma = linearToGamma rScaled
          gGamma = linearToGamma gScaled
          bGamma = linearToGamma bScaled

```

```haskell
          range = Interval 0.000 0.999
          ir = 256 * clamp range rGamma
          ig = 256 * clamp range gGamma
          ib = 256 * clamp range bGamma

        putStrLn $ show ir ++ " " ++ show ig ++ " " ++ show ib

writeColorStr :: Vec3 -> Int -> String
writeColorStr (Vec3 r g b) samples = res
        where
          scale = 1.0 / fromIntegral samples
          rScaled = r * scale
          gScaled = g * scale
          bScaled = b * scale

          rGamma = linearToGamma rScaled
          gGamma = linearToGamma gScaled
          bGamma = linearToGamma bScaled

          range = Interval 0.000 0.999
          ir = 256 * clamp range rGamma
          ig = 256 * clamp range gGamma
          ib = 256 * clamp range bGamma
          res  = show ir ++ " " ++ show ig ++ " " ++ show ib
```

src/Hittable.hs

```haskell
{-# LANGUAGE ExistentialQuantification #-}

module Hittable(
  Hittable,
  HittableList (HittableList),
  HitRecord (HitRecord),
  Material,
  Lambertian (Lambertian),
  Metal (Metal),
  Dielectric (Dielectric),
  scatter,
  hit,
  setFaceNormal
) where

import Vec3
import Ray
import Interval
import System.Random
```

```haskell

class Material a where
  scatter :: Ray -> HitRecord -> StdGen -> a ->  (Maybe (Ray, Vec3), StdGen)

data Lambertian = Lambertian Vec3
instance Material Lambertian where
  scatter _ (HitRecord p' n' _ _ _) g (Lambertian albedo) =
    (Just (Ray p' scatter_direction, albedo), g1)
      where
        (rand, g1) = randomUnitVector g
        new_scatter = n' `addVec3` rand
        scatter_direction = if (nearZero new_scatter) then n' else (new_scatter)

data Metal = Metal Vec3 Double
instance Material Metal where
  scatter ray (HitRecord p' n' _ _ _) g (Metal albedo fuzz) =
    if ((direction scattered) `dot` n' > 0) then (Just (scattered, albedo), g1) else (Nothing, g1)
      where
        reflected = reflect (unitVector (direction ray)) n'
        (rand, g1) = randomUnitVector g
        scattered = (Ray p' (reflected `addVec3` (rand `multiplyVec3` fuzz)))


data Dielectric = Dielectric Double
instance Material Dielectric where
    scatter ray (HitRecord p' n' _ _ f') g (Dielectric ir) =
        (Just (scattered, color), g1)
            where
                color = Vec3 1.0 1.0 1.0
                refractionRatio = if f' then 1.0 / ir else ir
                unitDirection = unitVector (direction ray)
                cosTheta = min (negateVec3 unitDirection `dot` n') 1.0
                sinTheta = sqrt (1.0 - cosTheta * cosTheta)
                cannotRefract = refractionRatio * sinTheta > 1.0
                (rd, g1) = randomDouble g
                scattered = if cannotRefract || reflectance cosTheta refractionRatio > rd then Ray p' (reflect unitDirection n') else Ray p' (refract unitDirection n' refractionRatio)

```

```

reflectance :: Double -> Double -> Double
reflectance cosine refIdx = ret
    where
        r0 = (1 - refIdx) / (1 + refIdx)
        r0' = r0 * r0
        ret = r0' * (1 + r0')*(1 - cosine)**5


data HitRecord = forall a. Material a => HitRecord Vec3 Vec3 a Double Bool

setFaceNormal :: Ray -> Vec3 -> HitRecord -> HitRecord
setFaceNormal r outward_normal (HitRecord pOriginal _ matOriginal tOriginal _) =
  HitRecord pOriginal new_normal matOriginal tOriginal new_front_face
  where
    new_front_face = direction r `dot` outward_normal < 0
    new_normal = if new_front_face then outward_normal else negateVec3 outward_normal

class Hittable a where
  hit :: Ray -> Interval -> Maybe HitRecord -> a -> Maybe HitRecord

newtype HittableList a = HittableList [a]

instance Hittable a => Hittable (HittableList a) where
  hit ray range record (HittableList items) = hitHelper ray range record (HittableList items)
    where
      hitHelper _ _ record' (HittableList []) = record'
      hitHelper ray' range' record' (HittableList (x':xs)) =
        case hit ray' range' record' x' of
              Nothing -> hitHelper ray' range' record' (HittableList xs)
              (Just valid@(HitRecord _ _ _ t' _)) -> hitHelper ray' (Interval (t_min range) t') (Just valid) (HittableList xs)
```

src/Interval.hs:

```
module Interval(
  Interval (Interval),
  contains,
```

```haskell
  surrounds,
  clamp,
  t_min,
  t_max,
) where

data Interval = Interval
  {
    t_min :: Double,
    t_max :: Double
  }

contains :: Double -> Interval -> Bool
contains x (Interval t_min1 t_max1) = x >= t_min1 && x <= t_max1

surrounds :: Double -> Interval -> Bool
surrounds x (Interval t_min1 t_max1) = t_min1 < x && x < t_max1

clamp :: Interval -> Double -> Double
clamp (Interval rMin rMax) val
        | val < rMin = rMin
        | val > rMax = rMax
        | otherwise = val
```

src/Lib.hs:

```haskell
module Lib
  ( someFunc
  ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

src/Ray.hs:

```haskell
module Ray(
  Ray (Ray),
  origin,
  direction,
  at
) where

import Vec3

data Ray = Ray
  {
    origin :: Vec3,
```

```
    direction :: Vec3
  } deriving (Show)

at :: Ray -> Double -> Vec3
at (Ray org dir) t = org `addVec3` (dir `multiplyVec3` t)
```

src/Sphere.hs:

```
{-# LANGUAGE ExistentialQuantification #-}

module Sphere(
  Sphere (Sphere)
) where

import Vec3
import Ray
import Hittable
import Interval
data Sphere = forall a. Material a => Sphere Vec3 Double a

instance Hittable Sphere where
  hit r range _ (Sphere cent rad mat) =
    let oc = origin r `minusVec3` cent
      a = lengthSquaredVec3 (direction r)
      half_b = oc `dot` (direction r)
      c = lengthSquaredVec3 oc - (rad * rad)

      discriminant = half_b * half_b - a * c

      checkRoot :: Double -> Bool
      checkRoot root = surrounds root range

      updateHitRecord :: Double -> HitRecord -> HitRecord
      updateHitRecord root (HitRecord _ _ mat2 _ f) =
        setFaceNormal r outward_normal (HitRecord hit_point outward_normal mat2 root f)
          where
            hit_point = at r root
            outward_normal = (hit_point `minusVec3` cent) `divideVec3` rad

    in if discriminant < 0
      then Nothing
      else
        let sqrtd = sqrt discriminant
          root1 = (-half_b - sqrtd) / a
```

```
          root2 = (-half_b + sqrtd) / a

          validRoot1 = checkRoot root1
          validRoot2 = checkRoot root2

        in case (validRoot1, validRoot2) of
          (True, _) -> Just $ updateHitRecord root1 (HitRecord (Vec3 0 0 0) (Vec3 0 0 0) mat 0 True)
          (_, True) -> Just $ updateHitRecord root2 (HitRecord (Vec3 0 0 0) (Vec3 0 0 0) mat 0 True)
          _ -> Nothing
```

src/Utilities.hs:

```
module Utilities(
  degreesToRadians,
  randomDouble,
  randomDoubleR
) where
import System.Random

degreesToRadians :: Double -> Double
degreesToRadians deg = deg * pi / 180.0

randomDouble :: StdGen -> (Double, StdGen)
randomDouble = randomR (0.0, 1.0)

randomDoubleR :: Double -> Double -> StdGen -> (Double, StdGen)
randomDoubleR rand_min rand_max gen =
  let (randValue, newGen) = randomDouble gen
  in (rand_min + (rand_max - rand_min) * randValue, newGen)
```

src/Vec3.hs:

```
module Vec3(
  Vec3 (Vec3),
  x,
  y,
  z,
  negateVec3,
  addVec3,
  minusVec3,
  multiplyVec3,
  divideVec3,
  lengthSquaredVec3,
  lengthVec3,
  unitVector,
```

```haskell
  dot,
  cross,
  randomVec3,
  randomUnitVector,
  nearZero,
  reflect,
  refract
) where

import System.Random
import Utilities

data Vec3 = Vec3
  {
    x :: Double,
    y :: Double,
    z :: Double
  } deriving (Show)

negateVec3 :: Vec3 -> Vec3
negateVec3 (Vec3 x' y' z') = Vec3 (-x') (-y') (-z')

addVec3 :: Vec3 -> Vec3 -> Vec3
addVec3 (Vec3 u1 u2 u3) (Vec3 v1 v2 v3) = Vec3 (u1 + v1) (u2 + v2) (u3 + v3)

class MultiplyVec3 a where
  multiplyVec3 :: Vec3 -> a -> Vec3

instance MultiplyVec3 Double where
  multiplyVec3 (Vec3 x' y' z') t = Vec3 (x' * t) (y' * t) (z' * t)

instance MultiplyVec3 Vec3 where
  multiplyVec3 (Vec3 a b c) (Vec3 x' y' z')  = Vec3 (a * x') (b * y') (c * z')

minusVec3 :: Vec3 -> Vec3 -> Vec3
minusVec3 (Vec3 u1 u2 u3) (Vec3 v1 v2 v3) = Vec3 (u1 - v1) (u2 - v2) (u3 - v3)

divideVec3 :: Vec3 -> Double -> Vec3
divideVec3 v t = multiplyVec3 v (1 / t)

lengthSquaredVec3 :: Vec3 -> Double
```

```haskell
lengthSquaredVec3 (Vec3 x' y' z') = x' * x' + y' * y' + z' * z'

lengthVec3 :: Vec3 -> Double
lengthVec3 v = sqrt (lengthSquaredVec3 v)

unitVector :: Vec3 -> Vec3
unitVector v = v `divideVec3` lengthVec3 v

dot :: Vec3 -> Vec3 -> Double
dot (Vec3 u1 u2 u3) (Vec3 v1 v2 v3) = (u1 * v1) + (u2 * v2) + (u3 * v3)

cross :: Vec3 -> Vec3 -> Vec3
cross (Vec3 a1 a2 a3) (Vec3 b1 b2 b3) = Vec3 (a2 * b3 - a3 * b2) (a3 * b1 - a1 * b3) (a1 * b2 - a2 * b1)

randomVec3 :: StdGen -> (Vec3, StdGen)
randomVec3 = randomVec3R 0.0 1.0

randomVec3R :: Double -> Double -> StdGen -> (Vec3, StdGen)
randomVec3R rand_min rand_max g =
  ((Vec3 x' y' z'), g3)
    where
      (x', g1) = randomDoubleR rand_min rand_max g
      (y', g2) = randomDoubleR rand_min rand_max g1
      (z', g3) = randomDoubleR rand_min rand_max g2

randomInUnitSphere :: StdGen -> (Vec3, StdGen)
randomInUnitSphere g
 | lengthSquaredVec3 v <= 1 = (v, g1)
 | otherwise = randomInUnitSphere g1
  where
    (v, g1) = randomVec3R (-1.0 ) 1.0 g

randomUnitVector :: StdGen -> (Vec3, StdGen)
randomUnitVector g = (unitVector v, g1)
  where
    (v, g1) = randomInUnitSphere g

nearZero :: Vec3 -> Bool
nearZero (Vec3 a b c) =
  (abs a < s) && (abs b < s) && (abs c < s)
    where s = 1e-8

reflect :: Vec3 -> Vec3 -> Vec3
```

```haskell
reflect v n = v `minusVec3` (n `multiplyVec3` (2 * (v `dot` n)))

refract :: Vec3 -> Vec3 -> Double -> Vec3
refract uv n refrac = rOutPerp `addVec3` rOutParallel
          where
            cosTheta = min (negateVec3 uv `dot` n) 1.0
            rOutPerp = (uv `addVec3` (n `multiplyVec3` cosTheta)) `multiplyVec3` refrac
            rOutParallel = n `multiplyVec3` ((-1) * sqrt (abs (1.0 - (lengthSquaredVec3 rOutPerp))))
```