Parallel Functional Programming Project: Parallel Ray Tracer

Garrison Grogan, UNI gg2652

December 17 2019

1 Introduction

I chose to write a very simple parallel ray tracer in Haskell. A ray tracer is a computer graphics program which, given a description of a scene, produces a photo realistic image through a ray tracing algorithm. The ray tracing algorithm in short shoots many rays from light sources in the scene, and calculates intersections between those rays and objects in the scene. Depending on the light source types implemented and material types implemented, this can produce a wide range of effects like reflections, refractions, global illumination, diffuse shading, etc.

I started with a reference description of a non paralleled ray tracer. I updated the code to follow more modern Haskell conventions, compile without warning on new versions of GHC, and made some changes changes to the parser, as well as the way rays were calculated. The full code can be found in the code listing section. It was compiled and tested using GHC 8.6.5. See the README in the included source code for full instructions on compiling and running the program. Use the included example scenes to see how to make new scene files.

The original code for calculating the rays was:

```
{- returns a color array representing the image -}
  getImage :: Int -> Resolution -> Scene -> [Color]
  getImage d r@(rx,ry) s = [image (fromIntegral x, fromIntegral (-y))
    | y<- [-(ry-1)..0], x <- [0..(rx-1)]]
      where
        image = rayTrace d r s
```

This is really the only paralellizable part of the code, since rays are independent of each other, and a single ray only has a few calculations done on it before it is complete. The non parallel version of the code actually worked nicely sequentially on my computer. On the example files I made they completed sequentially with the times (calculated from getCPUTime) in table 1. The sequential program does not scale at all though, and a bottle neck is reached when generating high resolution images. To see the example images used in testing, see figures 5-9 or generate them by running the program.

Example	Time (s)
1	5.9
2	3.2
3	5.9
4	12.2
5	8.4

Table 1: Sequential Runtime

When compiled with threading and event logging, the examples take significantly longer to generate. Example 4 now takes 19 to 20 seconds according to threadscope. This and all following runtime numbers come from looking at the total time variable in the event log using threadscope.

2 How to Parallelize?

From the above code, it seemed natural to want to split up the rays amongst cpu cores. The implementation already generates a list of ray drawing calls that return colors, so I first chose to use the evalList strategy talked about in class. I rewrote the list comphrension and it along with a parMap and rdeepseq call. I chose parMap since I already had an image function being applied to the list. I chose rdeepseq since the rays should be evaluated immediately. The only thing that happens to them after this generation step is being written to a file.

the results were pretty bad. See table 2 for the results on example 4, the most complex example image.

Example4 Time (s)	Core Count
19.04	1
18.12	2
16.55	3
17.54	4

Table 2: Example 4 parList Runtime

There was virtually no improvement. See Figure 1 for threadscope output on 4 cores. Clearly the cores are not being managed properly, only core 2 is being saturated. There is a bit of parallelization at the beginning of the run but it quickly stops. 1.9 million sparks were needlessly made and wasted on core 2. What went wrong is that the parallelization was too fine grained. Not every single ray needs to be executed in parallel. The computation cost of a ray is similar to that of a spark. Instead I should have been grouping the rays together in bundles.

I rewrote the the code to instead use parBuffer and rdeepseq. I split the the list of rays into 512 ray sized chunks, and evaluated each chunk in parallel instead of each ray. See the following code snippet:

```
{- returns a color array representing the image -}
getImage :: Int -> Resolution -> Scene -> [Color]
getImage d r@(rx,ry) s = concat $ withStrategy (parBuffer n rdeepseq)
  (map imagec $ chunks [(fromIntegral x, fromIntegral (-y)) |
  y<- [-(ry-1)..0], x <- [0..(rx-1)]])
```

```
where
  imagec x=  map (rayTrace d r s) x
  n = numCapabilities
  chunks x = chunksOf 512 x
```

This worked significantly better and I got a nice speed up. The single threaded case even gained a speed boost and was comparable to the non logging sequential code. See table 3 for the speed increases on example 4.

Example4 Time (s)	Core Count
12.76	1
7.39	2
5.4	3
4.5	4
4.26	5
4.01	6
4.09	7
4.15	8
3.92	9
3.85	10
3.75	11
4.09	12

Table 3: Example 4 parBuffer Runtime

There was diminishing return beyond 6 cores, which happens to be the physical core count on the computer I tested the program on. The computer does have 12 threads, and there was improvement of a quarter second until I fully saturated every thread and there was a speed loss. I am not sure why there was a brief slow down on 7 and 8 threads. It may have to do with the physical core count or just random tasks that were on those threads at runtime.

All tests had high efficiency according to threadscope, never dropping under 77 percent. Maximum efficiency was on 2 cores, at 91.3 percent, and slowly dropped. This is expected since the gains of adding cores will be marginal as more time is spent on communication between cores and reporting back when generating the final list. There were 0 overflowed sparks on any of the runs, and all cores maintained very similar spark counts. See figure 2,3,4 for threadscope results on 2,6 and 11 cores. Garbage collection became more of an issue above 10 cores but there was very little throughout the runs.

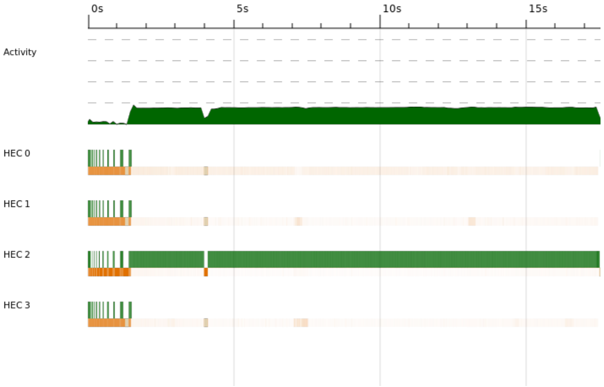


Figure 1: Threadscope Core Usage: Example 4, parList + rdeepseq

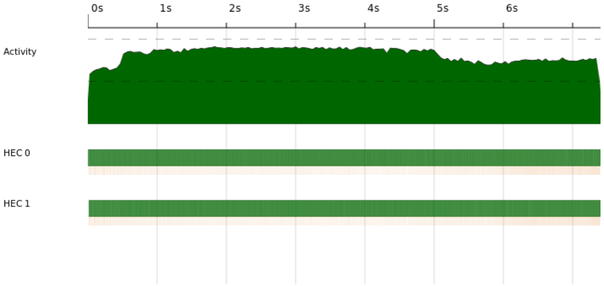


Figure 2: Threadscope - 2 Core Usage: Example 4, parBuffer + rdeepseq

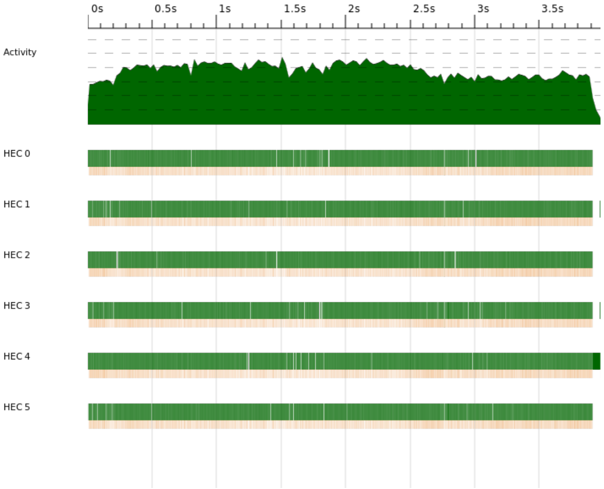


Figure 3: Threadscope - 6 Core Usage: Example 4, parBuffer + rdeepseq

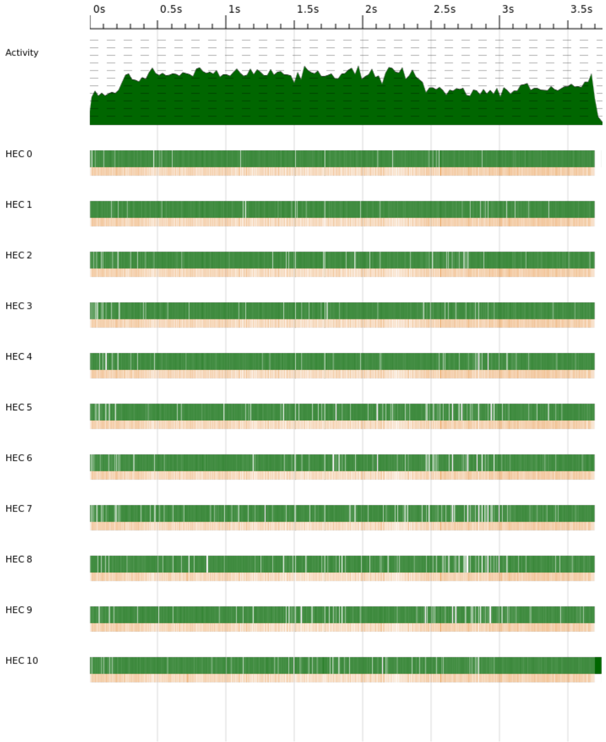


Figure 4: Threadscope - 11 Core Usage: Example 4, parBuffer + rdeepseq

3 Conclusion

Overall, there was pretty good 3x speed increase by efficiently parallizing the raytracer, though there were diminishing returns going beyond four cores, which became even more extreme after exceeding the testing computer's physical core count. There could still be some improvements though. The chunks are not selective as to what rays are difficult to calculate, and which are not. Difficult rays are likely next to each other, so some chunks are significantly harder to compute than others. There may be some load balancing solution which would remedy this. Furthermore the chunk size used in testing may actually be too small relative to spark creation cost still, it could be experimented with more. These improvements would need to be made as more features are added to the ray tracer like general polygon support, and more lighting and texture effects. The file format also needs to be improved. I used the ppm format since it was simple but the images should be generated in a more common format.

4 Example Images

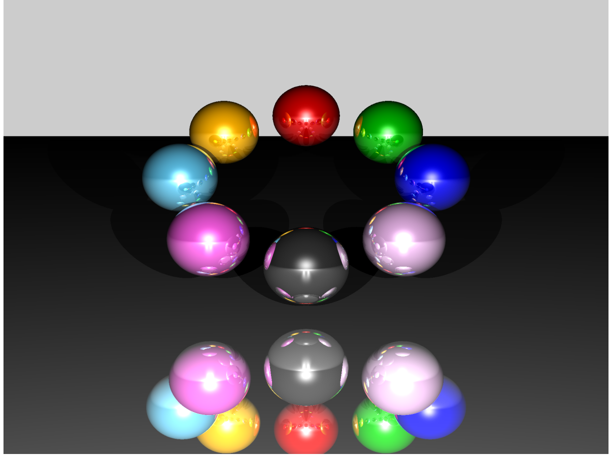


Figure 5: Example 1 Image

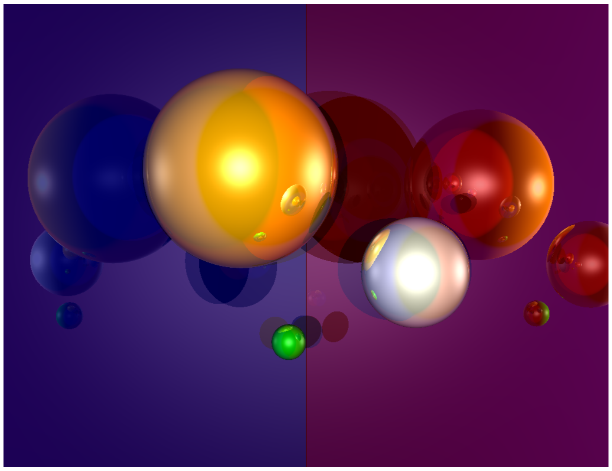


Figure 6: Example 2 Image

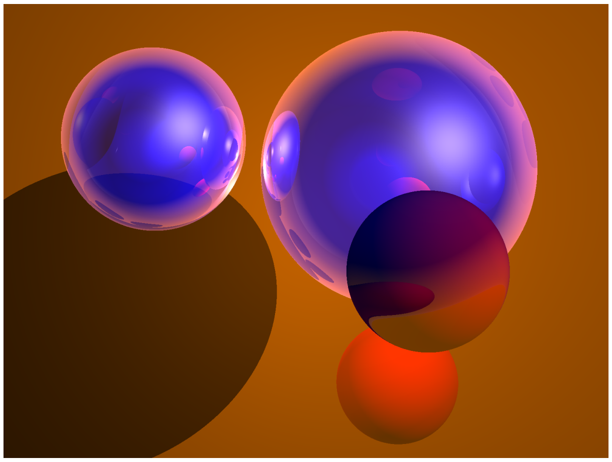


Figure 7: Example 3 Image

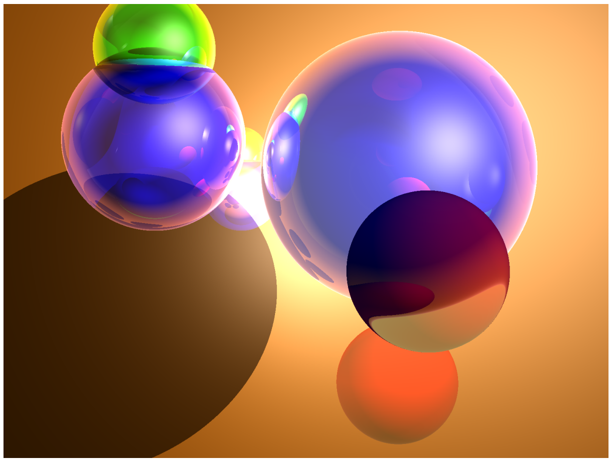
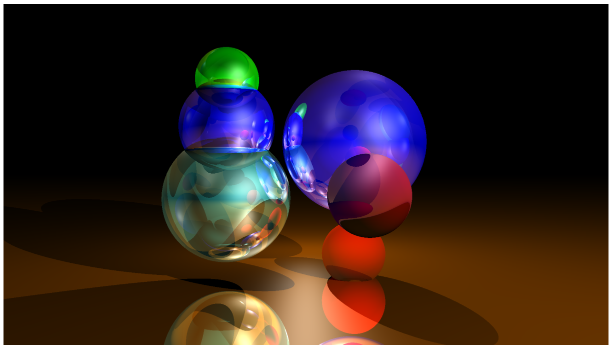


Figure 8: Example 4 Image



5 Code Listing

I have not included the generated parser code, since it over 800 lines long. I have included the yacc file.

```
import TraceOutput (getTime, createImage, getImage)
import RenParse (RendDesc(RendDesc), readScene)
import System.Environment (getArgs)
import System.CPUTime (getCPUTime)
import System.Directory (doesFileExist)

main :: IO()
main = do
  args <- getArgs
  if (length args /= 2) then
    error("Usage:␣Trace␣<scene␣path>␣<image␣path␣(ppm)>\n")
    else
      do
        let input = head args
        let output = head (tail args)
        exists <- doesFileExist input
        if (exists)
          then
            do content <- readFile input
               let (RendDesc res depth scene) = readScene content
               before <- getCPUTime
               createImage res (getImage depth res scene) output
               after <- getCPUTime
               putStr $ "Render␣Time:␣" ++ (getTime before after) ++ "\n"
          else error $ "File:␣\"" ++ input ++ "␣does␣not␣exist.\n"

module TraceOutput (getTime, createImage, getImage) where
  import Tracer (Resolution, Color, Scene, rayTrace)
  import System.IO (writeFile)
  import Control.Parallel.Strategies (withStrategy, parBuffer, rdeepseq)
  import GHC.Conc (numCapabilities)
  import Data.List.Split (chunksOf)

  getTime :: Integer -> Integer -> String
  getTime before after = show $ after - before

  {- returns a color array representing the image -}
  getImage :: Int -> Resolution -> Scene -> [Color]
  getImage d r@(rx,ry) s = concat $ withStrategy (parBuffer n rdeepseq) (map ima
    where
      imagec x=  map (rayTrace d r s) x
      n = numCapabilities
      chunks x = chunksOf 512 x


  {- makes a ppm file, easiest to make without librairies -}
  createImage :: Resolution -> [Color] -> String -> IO()
  createImage (w,h) colors name = do
      writeFile name str
        where
          str = ("P3\n"++) . shows w . (' ':) . shows h . ("\n255\n"++) . string
          stringify = flip $ foldr showC
          c = 255
          showC (r,g,b) = shows (round (r*c)::Integer)
            . (' ':) . shows (round (g*c)::Integer)
            . (' ':) . shows (round (b*c)::Integer) . (' ':)
```

[language=Haskell]

```
module RayMath where
  type Point2D = (Int, Int)
  type Point3D = (Double, Double, Double)
  type Vector = (Double, Double, Double)
  type Resolution = (Int, Int)

  type Dimension = (Int, Int) {-Screen window res if use GUI -}

  data Ray = Ray Point3D Vector
  data RenderObj = Sphere Double Point3D
                 | Plane (Double,Double,Double,Double)

  {- Didn't use folds, slower applying the fn -}
  (<+>) :: (Double, Double, Double) -> (Double, Double, Double) -> (Double, Doub
  (x1,y1,z1) <+> (x2,y2,z2) = (x1+x2, y1+y2, z1+z2)
  (<->) :: (Double, Double, Double) -> (Double, Double, Double) ->
(Double, Double, Double)
  (x1,y1,z1) <-> (x2,y2,z2) = (x1-x2, y1-y2, z1-z2)
  (<**>) :: (Double, Double, Double) -> (Double, Double, Double) -> (Double, Dou
  (x1,y1,z1) <**> (x2,y2,z2) = (x1*x2,y1*y2,z1*z2)

  (**>) :: (Double, Double, Double) -> Double -> (Double,Double,Double){-only us
  (x,y,z) **> f = (x*f,y*f, z*f)

  dot :: Vector -> Vector -> Double {- dot product -}
  dot (x1,y1,z1) (x2,y2,z2) = x1*x2 + y1*y2 + z1*z2

  len :: Vector -> Double
  len v = sqrt (v ‘dot‘ v)

  norm :: Vector -> Vector
  norm v
        | len v < 10**(-9) = (0.0,0.0,0.0) {- pesky floating point percision -}
        | otherwise = v **> (1/(len v))
  point2Vec :: Point3D -> Point3D -> Vector {- make a normalised vector from two
  point2Vec v w = norm (w <-> v)

  dist :: Point3D -> Point3D -> Double
  dist p0 p1 = sqrt ((p1 <-> p0) ‘dot‘ (p1 <-> p0))

  {-clipping the color to be in [0,1] -}
  clipUp :: Double -> (Double, Double, Double) -> (Double, Double, Double)
  clipUp f (x,y,z) = (max x f, max y f, max z f)
  clipDown :: Double -> (Double, Double, Double) -> (Double, Double, Double)
  clipDown f (x,y,z) = (min x f, min y f, min z f)
  colorClip :: (Double, Double, Double) -> (Double, Double, Double)
  colorClip = (clipUp 0.0) . (clipDown 1.0)

  instRay :: Point3D -> Point3D -> Ray
  instRay p1 p2 = Ray p1 (point2Vec p1 p2)

  solveQuad :: (Double,Double,Double) -> [Double] {-used for ray intersection -}
  solveQuad (a,b,c)
    | d<0 = []
    | d>0 = [(-b - sqrt d)/(2*a),(-b+sqrt d)/(2*a)]
    | otherwise = [-b/(2*a)]
    where
      d = b*b - 4*a*c

  rayIntersectWith :: Ray -> RenderObj -> [Double]
  {- solving for (x-cenX)^2 + (y-cenY)^2 + (z - CenZ)^2 = rad^2 -}
  rayIntersectWith (Ray start dir) (Sphere rad cent) = solveQuad (dir ‘dot‘ dir,
    where d = start <-> cent
  rayIntersectWith (Ray start dir) (Plane (a,b,c,d))
    | abs((a,b,c) ‘dot‘ dir) < 10**(-9) = []
    | otherwise = [- (d+((a,b,c) ‘dot‘ start) ) / ((a,b,c) ‘dot‘ dir)]

  normal :: Point3D -> RenderObj -> Vector
  normal p (Sphere rad cent) = norm ((p <-> cent) **> (1/rad))
  normal _ (Plane (a,b,c,_)) = norm (a,b,c)
  {- reflected direction given normalized direction and normal vectors -}
  reflectDir :: Vector -> Vector -> Vector
  reflectDir i n = i <-> (n **> (2*(n ‘dot‘ i)))
```

```
  {- refracted direction given normalized direction and normal vectors -}
  refractDir :: Vector -> Vector -> Double -> Vector
  refractDir i n r
    | v < 0 = (0.0,0.0,0.0)
    | otherwise = norm $ (i **> rc) <+> (n **> (rc*(abs c) - sqrt v))
    where
      c = n ‘dot‘ (i **> (-1))
      rc
        | c<0 = r -- if cosine < 0, inside sphere
        | otherwise = 1/r
      v = 1+(rc*rc) * (c*c -1)

  {- convert a pixel to a ray vector from camera eye -}
  resToWin :: Resolution -> Dimension -> Point2D -> Point3D
  resToWin (rx,ry) (w,h) (px,py) = (x/rxD, y/ryD, 0.0)
    where
      (rxD, ryD) = (fromIntegral rx, fromIntegral ry)
      (pxD, pyD) = (fromIntegral px, fromIntegral py)
      (wD, hD) = (fromIntegral w, fromIntegral h)
      (x, y) = ((pxD-rxD/2)*wD, (pyD-ryD/2)*hD)

module Tracer (Resolution, Color, Diffuse(Solid), Texture(Texture), TexturedObj,
where
  import Maybes
  import RayMath

  type Color = (Double, Double, Double)

  data Diffuse = Solid Color

  {- specularity reflectiveCoef specCoef refractiveIndex-}
  data Texture = Texture Diffuse Double Int Double Double

  type TexturedObj = (RenderObj, Texture)

  type Intensity = (Double, Double, Double)

  data Light = PointL Point3D Intensity
             | AmbientL Intensity

  {- fixed at 0,0,0, the point3d is the point camera looks at, dim is view windo
  data Camera = Camera Point3D Dimension

  data Scene = Scene Camera Color [TexturedObj] [Light]

  data Intersection = Intersection Double Ray TexturedObj
```

```
type Image = Point2D -> Color

intersectDist :: (Maybe Intersection) -> Double
intersectDist Nothing = 0.0
intersectDist (Just (Intersection d _ _)) = d

intersectText :: (Maybe Intersection) -> Texture
intersectText Nothing = Texture (Solid (0.0,0.0,0.0)) 0.0 0 0.0 0.0
intersectText ( Just (Intersection _ _ (_,t))) = t

intersectPnt :: (Maybe Intersection) -> Point3D
intersectPnt Nothing = (0.0,0.0,0.0)
intersectPnt ( Just (Intersection d (Ray start dir) _)) = start <+> (dir **> d

normalAt :: (Maybe Intersection) -> Vector
normalAt Nothing = (0.0,0.0,0.0)
normalAt i@(Just (Intersection _ _ (o, _ ))) = normal (intersectPnt i) o

colorAt :: (Maybe Intersection) -> Color
colorAt Nothing = (0.0,0.0,0.0)
colorAt (Just (Intersection _ _ (_,Texture (Solid color) _ _ _ _)))= color

{- closest intersection of a ray and an object, with distance > given -}
closeInter :: Ray -> (Maybe Intersection) -> TexturedObj -> (Maybe Intersectio
closeInter r i (o,m)
  | d > 10**(-6) && ((isNothing i) || d < (intersectDist i)) = Just (Intersect
  | otherwise = i
  where
    d = firstPos (rayIntersectWith r o)
    firstPos [] = 0.0
    firstPos (x:xs)
      | x > 10**(-6) = x
      | otherwise = firstPos xs

intersectList :: Ray -> [TexturedObj] -> (Maybe Intersection)
intersectList r o = foldl (closeInter r) Nothing o

{- diffuse color at intersection -}
diff :: (Maybe Intersection) -> Light -> Color
diff _ (AmbientL _) = (0.0,0.0,0.0)
diff i (PointL pos int) = (int **> ((point2Vec (intersectPnt i) pos) `dot` (no

{- specular color at intersection -}
spec :: (Maybe Intersection) -> Vector -> Light -> Color
spec _ _ (AmbientL _) = (0.0,0.0,0.0)
spec i d (PointL p int) = int **> (rCoef * ( ((normalAt i) `dot` h)**(fromInte
  where
    h = norm ((d **> (-1)) <+> (point2Vec (intersectPnt i) p))
    (Texture _ rCoef sCoef _ _) = intersectText i

shadePnt :: Intersection -> Vector -> [TexturedObj] -> Light -> Color
shadePnt _ _ _ (AmbientL int) = int --direction doesn't matter
shadePnt i d o l@(PointL pos _)
  | shadow = (0.0,0.0,0.0)
  | otherwise = (diff (Just i) l) <+> (spec (Just i) d l)
  where
    shadow = not (isNothing iShad) && (intersectDist iShad) <= dist (intersectP
    iShad = intersectList (instRay (intersectPnt (Just i)) pos) o

reflectPnt :: Int -> Intersection -> Vector -> [TexturedObj] -> [Light] -> Col
reflectPnt dep i d = colorPnt dep (Ray (intersectPnt (Just i)) (reflectDir d (

refractPnt :: Int -> Intersection -> Vector -> Color -> [TexturedObj] -> [Ligh
refractPnt dep i d b
  | rDir == (0.0,0.0,0.0) = (\_ _ -> (0.0,0.0,0.0))
  | otherwise = colorPnt dep (Ray (intersectPnt (Just i)) rDir) (b **> rCoef)
  where
    rDir = refractDir d (normalAt (Just i)) rInd
    (Texture _ _ _ rCoef rInd) = intersectText (Just i)

{- color in a point using all the color info -}
colorPnt :: Int -> Ray -> Color -> [TexturedObj] -> [Light] -> Color
colorPnt (-1) _ _ _ _ = (0.0,0.0,0.0)
colorPnt d r@(Ray _ dir) b objs l
  | isNothing i = b
  | otherwise = colorClip $ shadeCol <+> refCol <+> refrCol
  where
    shadeCol = foldl (<+>) (0.0,0.0,0.0) (map (shadePnt (fromJust i) dir objs)
    refCol
      | refCoef == 0.0 = (0.0,0.0,0.0)
      | otherwise = (reflectPnt (d-1) (fromJust i) dir objs l )**>refCoef
    refrCol
      | refrCoef == 0.0 = (0.0,0.0,0.0)
      | otherwise = (refractPnt (d-1) (fromJust i) dir b objs l)**>refrCoef
    i = intersectList r objs
    (Texture _ refCoef _ refrCoef _) = intersectText i

rayTracePnt :: Int -> Scene -> Point3D -> Color
rayTracePnt d (Scene (Camera lens _) a b c) p = colorPnt d (Ray p (point2Vec l

{-finally trace the scene, return list of colors repping the image-}
```

```
  rayTrace :: Int -> Resolution -> Scene -> Image
  rayTrace d r s@(Scene (Camera _ dim) _ _ _) = (rayTracePnt d s) . (resToWin r

{
module RenParse (RendDesc(RendDesc), readScene) where

import Tracer (Scene(Scene), Texture(Texture), Diffuse(Solid), TexturedObj,
                   Light(AmbientL, PointL), Camera(Camera))
import RayMath (Dimension, Resolution, RenderObj(Sphere,Plane))
import Data.Char (isSpace, isAlpha, isDigit)


--scene which should be rendered
data RendDesc = RendDesc Resolution Int Scene


readScene:: String -> RendDesc
readScene= parseScene.lexer

}

%name parseScene
%tokentype { Token }

%token
      int               { TokenInt $$ }
      double            { TokenDouble $$ }
      camera            { TokenCamera }
      background        { TokenBackground }
      diffuse           { TokenDiffuse }
      solid             { TokenSolid }
      texture           { TokenTexture }
      sphere            { TokenSphere }
      plane             { TokenPlane }
      obj               { TokenTexturedObj }
      objs              { TokenObjs }
      pointL            { TokenPointL }
      ambientL          { TokenAmbientL }
      lights            { TokenLights }
      scene             { TokenScene }
      resolution        { TokenResolution }
      rendDesc          { TokenRendDesc }
      '{'               { TokenOpenAcc }
      '}'               { TokenCloseAcc }
      '('               { TokenOpenBrack }
      ')'               { TokenCloseBrack }
```

```
        '['                 { TokenOpenHook }
        ']'                 { TokenCloseHook }
        ','                 { TokenComma }

%%

RendDesc : rendDesc Resolution int '[' Scene ']'
{ RendDesc $2 $3 $5}

Resolution   : '(' resolution '(' int ',' int ')' ')'
{ ($4,$6) }

Scene        : scene '{' Camera '}' '{' Background '}'
                     '{' objs Objs '}' '{' lights Lights '}'
{ Scene $3 $6 $10 $14 }

Camera       : camera '(' double ',' double ',' double ')'
                      '(' int ',' int ')'
{ Camera ($3,$5,$7) ($10,$12) }

Background   : background '(' double ',' double ',' double ')'
{ ($3,$5,$7) }

Obj      : sphere double '(' double ',' double ',' double ')'
{ Sphere $2 ($4,$6,$8) }
             | plane '(' double ',' double ',' double ',' double ')'
{ Plane ($3,$5,$7,$9) }

Diffuse        : solid '(' double ',' double ',' double ')'
{ Solid ($3,$5,$7) }

Texture     : '(' texture '(' diffuse Diffuse ')' double int double double')'
{ Texture $5 $7 $8 $9 $10 }

TexturedObj : '(' obj '(' Obj ')' Texture ')'
{ ($4,$6) }

Objs      : {- empty -}
{ [] }
             | Objs TexturedObj
{ $2 : $1 }

Light        : '(' pointL '(' double ',' double ',' double ')'
                          '(' double ',' double ',' double ')' ')'
{ PointL ($4,$6,$8) ($11,$13,$15) }
             | '(' ambientL '(' double ',' double ',' double ')' ')'
```

```
{ AmbientL ($4,$6,$8) }

Lights       : {- empty -}
{ [] }
             | Lights Light
{ $2 : $1 }

{

happyError :: [Token] -> a
happyError _ = error ”Parse error!!!”

data Token
       = TokenInt Int
       | TokenDouble Double
       | TokenCamera
       | TokenBackground
       | TokenDiffuse
       | TokenSolid
       | TokenTexture
       | TokenSphere
       | TokenPlane
       | TokenObjType
       | TokenTexturedObj
       | TokenObjs
       | TokenPointL
       | TokenAmbientL
       | TokenLight
       | TokenLights
       | TokenScene
       | TokenResolution
       | TokenRendDesc
       | TokenOpenAcc
       | TokenCloseAcc
       | TokenOpenBrack
       | TokenCloseBrack
       | TokenOpenHook
       | TokenCloseHook
       | TokenComma

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexNum (c:cs) 1
lexer ('{':cs) = TokenOpenAcc : lexer cs
lexer ('}':cs) = TokenCloseAcc : lexer cs
lexer ('(':cs) = TokenOpenBrack : lexer cs
lexer (')':cs) = TokenCloseBrack : lexer cs
lexer ('[':cs) = TokenOpenHook : lexer cs
lexer (']':cs) = TokenCloseHook : lexer cs
lexer (',':cs) = TokenComma : lexer cs
lexer ('-':cs) = lexNum cs (-1)

lexNum cs mul
          | (r == '.')  = TokenDouble (mul * (read (num++[r]++num2) :: Double))
          | otherwise = TokenInt (round (mul * (read num))) : lexer (r:rest)
          where (num,(r:rest)) = span isDigit cs
                (num2,rest2)   = span isDigit rest

lexVar cs =
   case span isAlpha cs of
      ("camera",rest)             -> TokenCamera : lexer rest
      ("background",rest)         -> TokenBackground : lexer rest
      ("diffuse",rest)            -> TokenDiffuse : lexer rest
      ("solid",rest)              -> TokenSolid : lexer rest
      ("texture",rest)            -> TokenTexture : lexer rest
      ("sphere",rest)             -> TokenSphere : lexer rest
      ("plane",rest)              -> TokenPlane : lexer rest
      ("obj",rest)             -> TokenTexturedObj : lexer rest
      ("objs",rest)            -> TokenObjs : lexer rest
      ("pointL",rest)         -> TokenPointL : lexer rest
      ("ambientL",rest)           -> TokenAmbientL : lexer rest
      ("light",rest)              -> TokenLight : lexer rest
      ("lights",rest)             -> TokenLights : lexer rest
      ("scene",rest)              -> TokenScene : lexer rest
      ("resolution",rest)         -> TokenResolution : lexer rest
      ("rendDesc",rest)           -> TokenRendDesc : lexer rest
  }
```