The Monad.Reader/Issue5/HRay: A Haskell ray tracer

From HaskellWiki

< The Monad.Reader | Issue5(Redirected from The Monad.Reader/Issue5/HRay:A Haskell ray tracer)

1 HRay: A Haskell ray tracer

by Kenneth Hoste for The Monad.Reader Issue Five, 2005/06/11

As a thesis subject for Ghent University, I chose to write a ray tracer (HRay) in Haskell. The goal was to show how elegant, short and maintainable a ray tracing implementation would be in a functional language, as opposed to an imperative or procedural language. To achieve that goal, I first created a formal model for the application, using the functional and declarative formalism Funmath<ref>R.T. Boute "Concrete Generic Functionals: Principles, Design and Applications". Generic Programming 89-119, Kluwer, 2003</ref>. In a next phase, the model was implemented in Haskell to create a working version of the ray tracer. Additional features were added to the model, including a command line and graphical user interface, a parser for input specifications and neat texture support using the Perlin Noise technique.

This article describes the Haskell implementation of HRay. For more information about the formal model and my thesis, check the HRay website at [1] (http://www.elis.ugent.be/~kehoste/Haskell/HRay) .

2 Contents

Contents

2.1 Overview

Before discussing the actual implementation, a brief description of the ray tracing algorithm is given. The implementation is split up into several modules, which are discussed one by one. First of all, the necessary mathematical functions to support the ray tracing algorithm are discussed. Based on that module, functions are presented which implement the ray tracer. To support some extra textures, a Perlin noise module is implemented, which will be discussed briefly. The two user interfaces are presented, one using command line arguments and one using the Haskell GUI library Gtk2Hs. Finally, some conclusions are drawn.

2.2 Ray tracing

The powerful ray tracing algorithm is well known for the amazing results it produces<ref>Hall of Fame (http://www.povray.org/community/hof/) </ref>. Because the rest of the article is based on the algorithm, we'll give a brief description of it. If you want more details, a good place to start is <ref name="raytrace2-3">M. Slater, A. Steed and Y. Chrysanthou. *Computer Graphics and Virtual Environments - From Realism to Real-Time*. Addison-Wesley, 2001</ref>.

As the name states, the ray tracing algorithm is built upon the notion of rays. A ray is constructed through the point of view and some point of the view window. Given a desired resolution and collection of 3D objects, we calculate the intersection of the ray with the objects, for each pixel of the image. Again using rays, which are constructed as needed, the color for the intersection point (if any) is determined, depending on the color (or texture) of the object and the surrounding objects and light sources. Iterating over all pixels of the image results in a matrix of colors, representing the image.

In pseudo-code:

```
for each pixel
 construct ray from view point to pixel location
(*) determine closest intersection of ray with objects
 if no intersection
 pixel color = background color
 else
 for each light source
 construct ray from intersection point to light source location
 if intersection found between intersection point and light source
 shadow, so color = black
 else
 determine color contribution for light source, remember it
 if reflective surface
 construct reflected ray, determine contribution (*)
 if transparent surface
 construct refracted ray, determine contribution (*)
 determine color for pixel by summing all contributions
```

This is all quite imperative. The next step we need is to construct functions which can help with implementing the algorithm using a functional style. This is where the formal model came into play, but because we want to concentrate on the implementation of it all in Haskell, I'll just use the model informally: we'll first construct some basic math functions before moving on to the actual ray tracing engine. The needed types and datatypes are presented as needed.

2.3 Math functions

2.3.1 Types and datatypes

First of all, we mention the several (data)types needed to support the needed basic math functions. Because we are making a transition from a 2D to a 3D world, we need to be able to represent points in both worlds. Also, we need vectors for direction purposes, which are defined the same way as points in the 3D world are. This was a little point of critique in the formal model, but because in Haskell giving objects two different types is sufficient to distinguish between them, there is no problem. On the contrary, this way of defining the types supports overloading of functions such as multiplying a point/vector with some factor, and calculating the sum of two points/vectors.

```
type Point2D = (Int,Int)
type Point3D = (Double,Double,Double)
type Vector = (Double,Double,Double)
```

Using these new types, we can define a Ray datatype as follows.

```
data Ray = Ray Point3D Vector
```

Intuitively, every ray has a point of origin and a direction. Besides rays, the supported objects are also important in this module. For this purpose, the datatype defining the several supported objects is also defined here. To keep the implementation simple, we only support spheres and planes.

```
data Object = Sphere Double Point3D
 | Plane (Double,Double,Double,Double)
```

For type-convenience and clarity, we also introduce a Resolution and a Dimension type. The use of these types will become clear later on.

```
type Resolution = (Int,Int)
type Dimension = (Int,Int)
```

2.3.2 Basic math functions

When modelling the ray tracer, the need arose for the extension of some elementary operations on points and vectors. The functions supporting these operations where defined as infix operators, so the several expressions wouldn't become gibberish (don't you just hate when that happens in imperative languages). The first of these operations (sum of two points/vectors, subtraction of two points/vectors, multiplication of two points/vectors and multiplication of a point/vector with a factor) are given below. For efficiency, the types were not made polymorphic (i.e. using Num).

```
(<+>) :: (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,Double,Double)
(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)
(x,y,z) *> f = (x*f,y*f,z*f)}
```

Also the classic max and min operators are extended, to make the definition of a clip function (which will adjust the RGB values of colors) really simple.

```
maxF :: Double -> (Double,Double,Double) -> (Double,Double,Double)
maxF f (x,y,z) = (max x f, max y f, max z f)

minF :: Double -> (Double,Double,Double) -> (Double,Double,Double)
minF f (x,y,z) = (min x f, min y f, min z f)
```

Besides the generalised functions above, we have defined elementary vector-only functions: scalar product of two vectors, length of a vector, normalizing a vector and creating a normalized vector given two points.

```
(*.) :: Vector -> Vector -> Double
(x1,y1,z1) *. (x2,y2,z2) = x1*x2 + y1*y2 + z1*z2

len :: Vector -> Double
len v = sqrt (v *. v)

norm :: Vector -> Vector
norm v
 | len v < 10**(-9) = (0.0,0.0,0.0)
 | otherwise = v *> (1/(len v))

mkNormVect :: Point3D -> Point3D -> Vector
mkNormVect v w = norm (w <-> v)
```

For shadow purposes, we need a function which determines the distance between two given points, and to avoid 'color spilling', where the RGB values of a color reach values outside the [0,1] interval, we define a clipping function.

```
dist :: Point3D -> Point3D -> Double
dist p0 p1 = sqrt ((p1 <-> p0) *. (p1 <-> p0))

clip :: (Double,Double,Double) -> (Double,Double,Double)
clip = (maxF 0.0) . (minF 1.0)
```

2.3.3 Geometric and other math functions

Several functions which deal with the geometric issues involved in playing around with rays and geometric objects, are defined to support the ray tracing calculations. An important function is a quadratic solver, which is kept simple: it just returns a list of results... This way, it is compatible with future functions, which might produce more solutions.

```
solveq :: (Double,Double,Double) ->[Double]
solveq (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
```

For ray construction, we already have the data constructor Ray which comes with the data declaration of Ray. Because we also will need to create a ray given two points, we define a function which creates a normalized vector out of these points, and uses the Ray data constructor internally.

```
mkRay :: Point3D -> Point3D -> Ray
mkRay p1 p2 = Ray p1 (mkNormVect p1 p2)
```

To determine the intersection of a ray with some object, we provide the intRayWith function. It returns a collection of intersection points, which should be sorted from close to far. Since we only support spheres and planes, the code needed if fairly simple.

```
intRayWith :: Ray -> Object -> [Double]
intRayWith (Ray start dir) (Sphere rad cen) = solveq (dir *. dir, 2*(dir *. d), (d *. d) - rad^2)
 where
 d = start <-> cen
intRayWith (Ray start dir) (Plane (a,b,c,d)) = if (abs(part) < 10**(-9)) then [] else [- (d + ((a,b,c) *. start)
 where
 part = (a,b,c) *. dir
```

Ray tracing accurately models light information, i.e. shadows, highlights and fading colors. The calculations which make this possible, use the normals on the surfaces of objects to determine the angle in which the object is seen. For that purpose, we support a normal function (again, fairly simple).

```
normal :: Point3D -> Object -> Vector
normal p (Sphere rad cen) = norm ((p <-> cen) *> (1/rad))
normal _ (Plane (a,b,c,d)) = norm (a,b,c)
```

The real power of the ray tracing algorithm is the ease in which it models real-life effects such a reflection and refraction of light rays. Giving some direction, we provide functions to determine the reflected and refracted directions when the incoming ray intersects with some surface.

```
reflectDir :: Vector -> Vector -> Vector
reflectDir i n = i <-> (n *> (2*(n *. i)))
```

```
refractDir :: Vector -> Vector -> Double -> Vector
refractDir i n r = if (v < 0) then (0.0, 0.0, 0.0) else norm $ (i *> r_c) <+> (n *> (r_c*(abs c) - sqrt v))
 where
 c = n *. (i *> (-1))
 r_c = if (c < 0) then r else 1/r -- when cosVal < 0, inside of sphere (so travelling to vacuum)
 v = 1 + (r_c^2) * (c^2 - 1)
```

Because the user can choose the resulting image resolution and the size of the view window, we need to map the pixel indices onto the window dynamically. This is done using the mapToWin function below.

```
mapToWin :: Resolution -> Dimension -> Point2D -> Point3D
mapToWin (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 )
```

2.4 Ray tracing engine

Using the mathematical basis of the HRayMath module, we can define the functions which will support the actual raytracing. First, we explain the needed (data)types, in the same way as we did with the math module. Next, we provide some functions which allow extraction of the needed information from an instance of the Intersection datatype. We discuss the functions which do the actual intersecting of rays with objects, determine the color of a pixel and support reflection and refraction effects. At the end of this section, we show the core functions of the raytracing engine, which are pretty short and elegant.

2.4.1 Types and datatypes

When working with (3D) graphics, the notion of color is essential. We represent a color as an RGB value (values between 0 and 1).

```
type Color = (Double,Double,Double)
```

As already mentioned, we will support special textures for all objects, besides just plain colors. For this purpose, we created a special datatype called Diff (diffuse color).

```
data Diff = Solid Color |
 Perlin (Point3D -> Color)
```

Each texture, whether plain or Perlin, has several parameters, to determine the reflection and refraction effects of the specific material. We support specular coefficient along with specularity, and refraction coefficient along with refraction index.

```
data Texture = Texture Diff Double Int Double Double
```

Every object in our scene is has a certain type (sphere, plane) and a texture attached to it. Hence we provide a TexturedObject datatype.

```
type TexturedObject = (Object,Texture)
```

Every light source has a certain color, which we call Intensity. That way one can use red, blue, bright and dark lights. We also support 2 basic types of light sources: ambient lights, which is actually a correcting factor, and point lights, which have a certain position in the 3D space.

```
type Intensity = (Double,Double,Double)

data Light = PointLight Point3D Intensity
 | AmbientLight Intensity
```

The point of view (or camera) determines which view is rendered of the provided scene. We allow to define the point in space through which the scene is looked at, and the view window dimensions. The view window is always located in the XY-plane in the origin of the axis.

```
data Camera = Camera Point3D Dimension
```

The whole scene is described by describing the camera, background color, list of objects and list of light sources.

```
data Scene = Scene Camera Color [TexturedObject] [Light]
```

An intersection of a ray with an object provides the necessary information for the raytracing calculations. We include the distance to the intersection, the intersecting ray and the object which is intersected.

```
data Intersection = Intersection Double Ray TexturedObject
```

The final product of the algorithm, an image representing the 3D scene, is actually a matrix of color. That way, each pixel pair is mapped to a RGB color tuple.

```
type Image = Point2D -> Color
```

2.4.2 Intersection functions

Because not every ray will intersect with an object, we use the Maybe monad in combination with our own Intersection datatype. In order to make working with the artificial (Maybe Intersection) type easier, we provide some functions which extract information from a possible intersection. This design choice might be an opportunity for improvement. We show the distance, texture, color, normal and intersection point functions. Notice how the color of some point in the 3D space is dependent on the location for Perlin textures.

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

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

```
colorAt :: (Maybe Intersection) -> Color
colorAt Nothing = (0.0,0.0,0.0)
colorAt (Just (Intersection _ _ (_,Texture (Solid color) _ _ _ _) )) = color
colorAt i@(Just (Intersection _ _ (_,Texture (Perlin f) _ _ _ _) )) = f (intPt i)
```

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

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

2.4.3 Intersecting

To determine the intersection (if any) of a ray with the collection of objects, we use the functions below.

At first, a simple function is used to determine the first positive element in a list of doubles. We chose not to remove the negative values from the list of intersection distances because we might need them for other uses, when extending the features of the ray tracer.

```
fstPos :: [Double] -> Double
fstPos [] = 0.0
fstPos (l:ls) = if l > 10**(-6) then l else fstPos ls
```

This function determines whether the intersection formed by the given ray and object (if any) is better than the known intersection.

```
closestInt :: Ray -> (Maybe Intersection) -> TexturedObject -> (Maybe Intersection)
closestInt r i (o,m) = if d > 10**(-6) && ((isNothing i) || d < (intDist i))
 then Just (Intersection d r (o,m))
 else i
 where
 d = fstPos (intRayWith r o)
```

To intersect a ray with a collection of objects, we can use the powerful foldl operator in combination with the closestInt function (which is given a basic empty intersection to start with).

```
intersect :: Ray -> [TexturedObject] -> (Maybe Intersection)
intersect r o = foldl (closestInt r) Nothing o
```

2.4.4 Determining color

The color of an intersection point is determined by several factors: the diffuse color of the object itself (and the angle of view), the specular influence of the light sources and the other objects, which might be blocking the light from some light sources. The functions diff, spec and shadePt do the necessary calculations to determine the color at a given intersection point. For more details on how this is done, we point to<ref name="raytrace3-2"/>.

```
diff :: (Maybe Intersection) -> Light -> Color
diff _ (AmbientLight _) = (0.0,0.0,0.0)
diff i (PointLight pos int) = (int *> ((mkNormVect (intPt i) pos) *. (normalAt i))) <*> (colorAt i)

spec :: (Maybe Intersection) -> Vector -> Light -> Color
spec _ _ (AmbientLight _) = (0.0,0.0,0.0)
spec i d (PointLight pos int) = int *> (reflCoef * ( ((normalAt i) *. h)**(fromIntegral specCoef) ))
 where
 h = norm ((d *> (-1)) <+> (mkNormVect (intPt i) pos))
 (Texture _ reflCoef specCoef _ _) = intText i

shadePt :: Intersection -> Vector -> [TexturedObject] -> Light -> Color
shadePt i d o (AmbientLight int) = int
shadePt i d o l@(PointLight pos int)
 | s = (0.0,0.0,0.0)
 | otherwise = (diff (Just i) l) <+> (spec (Just i) d l)
 where
 s = not (isNothing i_s) && (intDist i_s) <= dist (intPt (Just i)) pos
 i_s = intersect (mkRay (intPt (Just i)) pos) o
```

2.4.5 Recursive functions

The reflection and refraction effects are basically contributions of the other objects in the scene to the color of an intersection point. In the raytracing algorithm, this is modelled pretty easily: just look at the other objects in the scene from the intersection, and add the color observed in that point to the color of the intersection (of course taking into account the reflection/refraction component of the texture of the object). In other words: use recursion.

```
reflectPt :: Int -> Intersection -> Vector -> [TexturedObject] -> [Light] -> Color
reflectPt depth i d = colorPt depth (Ray (intPt (Just i)) (reflectDir d (normalAt (Just i)))) (0.0,0.0,0.0)

refractPt :: Int -> Intersection -> Vector -> Color -> [TexturedObject] -> [Light] -> Color
refractPt depth i d b = if refractedDir == (0.0,0.0,0.0) then (\x y -> (0.0,0.0,0.0))
 else colorPt depth (Ray (intPt (Just i)) refractedDir) (b *> refrCoef)
 where
 refractedDir = refractDir d (normalAt (Just i)) refrIndex
 (Texture _ _ _ refrCoef refrIndex) = intText (Just i)
```

2.4.6 Core functions

To determine the actual color at some pixel of the image, we shoot a ray through the pixel from some point and add all the different components (object, reflection, refraction) together. To avoid infinite recursion (rays bouncing between two point without changing direction), we use a recursion depth parameter.

```
colorPt :: Int -> Ray -> Color -> [TexturedObject] -> [Light] -> Color
colorPt (-1) _ _ _ _ = (0.0, 0.0, 0.0)
colorPt d r@(Ray _ dir) b o l = if (isNothing i) then b else clip $ shadeColor <+> reflectColor <+> refractColor
 where
 shadeColor = foldl (<+>) (0.0,0.0,0.0) (map (shadePt (fromJust i) dir o) l)
 reflectColor = if (reflCoef == 0.0) then (0.0, 0.0, 0.0)
 else (reflectPt (d-1) (fromJust i) dir o l) *> reflCoef
 refractColor = if (refrCoef == 0.0) then (0.0, 0.0, 0.0)
 else (refractPt (d-1) (fromJust i) dir b o l) *> refrCoef
 i = intersect r o
 (Texture _ reflCoef _ refrCoef _) = intText i
```

The actual algorithm translation to Haskell is really simple: for every pixel, map it to the view window, create a ray and determine the color.

```
rayTracePt :: Int -> Scene -> Point3D -> Color
rayTracePt d (Scene (Camera eye _) b o l) p = colorPt d (Ray p (mkNormVect eye p)) b o l

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

2.5 Perlin noise textures

Because the article would be too long, we don't elaborate on the implementation of the Perlin Noise textures. The code is freely available on http://scannedinavian.org /~boegel/HRay (links is dead), together with a gallery of images produced by the raytracer. If you want more information on the technique, please contact me (again, see the website).

2.6 A scene description parser

To support creation of scenes, we created a small, simple parser using Happy. When downloading the source code from the website, several example files are included, which should be enough to create your own scenes. Again, because of space limitations, we won't elaborate on the parser implementation.

2.7 User interfaces

To simplify usage of the raytracer, two simple interfaces were created. First, a very basic command line interface is provided, which gives limited feedback (render time, parse errors). Besides that, a simple GUI was created using the Gtk2Hs graphical library, a Gtk binding for Haskell. This provides a bit more feedback, but requires that Gtk2Hs installed. For more information on Gtk2Hs, one should read my article published in the first issue of The Monad.Reader (Media:TMR-Issue1.pdf).

2.8 Performance issues

While implementing the raytracer, performance was no issue. The goal was not to implement to fastest possible raytracer imaginable, but to show how elegant and clear a Haskell implementation can be. Because of this, a lot of improvements can be made to the existing code. Everyone who feels like doing so is free to use the code and make the necessary adjustments.

2.9 Conclusions

Recently, Jean-Philippe Bernardy added metaball support to HRay (check the gallery at http://scannedinavian.org/~boegel/HRay for images). While implementing, we used the Haskell98 standard. Several improvements are possible, including using type classes, which wasn't possible without using extensions to the Haskell98 standard (more specifically, existential types). Contributions, remarks or/and bugs are welcome. Have fun rendering and hacking!

2.10 References

<references/>

Retrieved from "http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue5/HRay:_A_Haskell_ray_tracer"

Category: Article
