Ray Tracing in One Weekend - Haskell Implementation

This project is a Haskell implementation of the ray tracer described in Ray Tracing in One Weekend by Peter Shirley. It demonstrates the translation of object-oriented concepts into functional programming paradigms.

Overview

This project stemmed from me wanting to gain a better understanding of Haskell throughout this course, as well as my interest i computer graphics. I came acreoss the tutorial for building a ray tracer in a weekend and thought it would be interesting to learn about rendering, graphics, and their applications, as well as serve as a launching point for me to dig into Haskell deeper. Throughout this I wanted to have completed the tutorial, in Hasekll instead of C++ as it was written, learn more about Haskell and graphics in the process, and have an output image that matched the tutorial. Throughout this project I was able to use, learn, and implement code in Haskell, a functional language, that maintains state, handles side effects, and ALMOST produces the correct final output. You can see more about my learning in the sections below. ## Implementation

Major Tasks and Capabilities

1. Conversion of Object-Oriented Programming (OOP) concepts to Functional Programming paradigms
2. Translation from C++ to Haskell, leveraging Haskell's strong type system and pure functions
3. Implementation of stateful computations using the IO monad to handle side effects
4. Modular scene rendering with customizable spheres, materials, and camera positions
5. Adjustable ray bounces and antialiasing for fine-tuning render quality

Components

The project is structured around several key modules:

1. **Camera.hs**: Defines the `Camera` data type and handles the rendering process. It uses the IO monad for rendering and writing the output image.

2. **Hittable.hs**: Implements the `Hittable` typeclass and `Materials` typeclass, defining how objects interact with rays. It includes implementations for Lambertian, Metal, and Dielectric materials.

3. **Sphere.hs**: Implements the `Sphere` data type and its instance of the `Hittable` typeclass.

4. **Ray.hs**: Defines the `Ray` data type and functions for working with rays.

5. **Vec3.hs**: Implements vector operations for 3D points and colors (not provided in the snippets, but referenced).

Functional Programming Highlights

1. Use of typeclasses (`Hittable`, `Materials`) to achieve polymorphism in a functional context
2. Extensive use of the IO monad for handling random number generation and file output
3. Pure functions for core computations
4. Pattern matching and case expressions for control flow (e.g., in the `hit` function for Sphere)

Project Status

Working Features

1. Full implementation of basic ray tracing with spheres
2. Multiple material types (Lambertian, Metal, Dielectric)
3. Adjustable camera with depth of field
4. Antialiasing and adjustable sample count

Partially Working Features

1. I seem to have an error in the Dielectrics which makes the "glass"/`Dielectric Spheres` render improperly.

Unimplemented Features

1. Advanced features from the follow-up books in the series

Tests

While specific test files weren't provided, the code structure allows for:

1. **Resolution and Quality Tests**: By adjusting `samplesPerPixel` and `maxDepth` in the `Camera` data type.
2. **Camera Positioning Tests**: Through modification of `lookFrom`, `lookAt`, and `vFOV` in the `Camera` data type.
3. **Material Tests**: By creating scene with different combinations of Lambertian, Metal, and Dielectric materials.
4. **Spheres Tests**: By creating scene with `Spheres` of different sizes and existing in different 3D spaces

These tests exercise key concepts such as:

1. The effectiveness of the IO monad in handling random number generation with Ray bounces within `Dielectrics`

2. The correctness of intersection calculations in the `Hittable` typeclass implementations
3. The accuracy of vector mathematics (implied by the use of Vec3 operations)

Listing

The code is commented so you should be able to see the ideas behind data structures and flow used, but a short list of the files and their implementations and importances are listed in this section

Camera.hs

Basic ray tracing camera and rendering system for basic 3D scenes (Main render loop)

1. Camera Data Structure: Represents a camera which contains many attributes similar to camera's in the real world (aspect ratio, field of view, placement of it in a 3d scene, etc)
2. Ray Tracing Functions: `rayColor` (calculates the color seen along a given ray), `getRay` zzzzz9generates a ray for a specific pixel, and add antialiasing and defocus effects)
3. Data Flow: `initialize` camera settings -> render scene based on `rayColor` of the rays originating from the camera -> calls `writeColor` and saves an output `.ppm` image.

Key Takeaways:

1. Use of iterating and accumulating results within the IO Monad used to handle state and the resulting side effects
2. Haskell has lazy evaluation, which I thought should make the program more efficient than the C++ implementation, but for some reason it seems to not be the case, potentially due to other overheads like garbage collection, and the data immutability meaning we have a lot of temporary data

Color.hs

Defining the `Color` data type, which is just a wrapper over `Vec3`. Also provides function for writing each pixel to file `writeColor`, and a function for gamma color correction `linearToGamma`

Hittable.hs

Provides the infrastructure to define the behavior of objects in the seen which can be `hit` or intersected with a `Ray`, and how those `Rays` interact based on an object's `Material`

1. HitRecord Data Strucutre: A record holding details about an object-ray intersection. This includes the point `p` at which the intersection occurs,

the surface `normal` at `p`, the `Material (mat)` of the object, the parameter `t` along the ray, and whether ot not the intersection occured on the `frontFace`
2. Hittable Type Class: Defines the `hit` function that determines an object-ray intersection and returns a `HitRecord` for any. Each `Hittable` object then can define how `Rays` interact with each of them individually.
3. Materials Type Class: Defins the `scatter` function that determines how a `Ray` shoudl interact with different materials and the direction they should aim after an intersection (produces a new `Ray` as well as giving it a `Color`)
4. Material Data Type: Wrapper that enaples storing and working with objects that implement `Materials` type class. Implementation of polymorphism in functional programming.
5. Specific Materials (Lambertion/Metal/Dielectric): Implements the `Materials` type class in order to define the `scatter` functionality for them.

Key Takeaways:

1. Polymorphism in functional programming through the use of wrappers in data types using the syntax `data DataType = forall t. (TypeClass t) => DataType t`
2. Type classes for the ability to define generic interfaces which different objects and materials can implement in unique ways

HittableList.hs

Define the data type HittableList, as well as helper functions to add to the list and clear it.

1. HittableObject Data Type: A wrapper for the Type Class `Hittable`, which like the `Material` Data Type, allows for us to be able to store and interact with objects which are `Hittable`
2. HittableList Data Type: Is a list of `HittableObjects` which can be cleared using `clearHittableList` or add a `HittableObject` using `addToHittableList`

Interval.hs

Defines a data type `Interval` which is a range between a minimum `iMin` and a maximum `iMax` as well as helper functions for getting the size, seeing if a number is within the range, and clamping output based on a range.

Ray.hs

Defines the data type for a `Ray` which contains its `origin` and the `direction` in which it is pointed

Sphere.hs

Defines the data type for A `Sphere` which implements the `hit` function as it is an instance of the type class `Hittable`. A `Sphere` is defined by is center point `sphereCenter`, the radius `sphereRadius`, and the `Material` it is made of `sphereMat`.

Vec3.hs

The underlying data type for `Color` and `Point3` are defined here. A `Vec3` is a 3D vector containing an `x`, `y`, and `z` components (can be implemented as `r`, `g`, and `b` components in the cas of `Color`). This also is where all of the functionality of how `Vec3` math shoul doperate is defined with making the components of `Vec3` be an instance of the built-in `Num` type class as well as defining `dot` product, `cross` product, and more.

VectorConstants.hs

This is more of a Utilities.hs as it contains useful constants and helper functions which do not necessarily fit within the other components and thus are in a seperate file

Acknowledgements

1. Thanks to Peter Shirley for the excellent “Ray Tracing in One Weekend” tutorial.