An Elixir ray tracer

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Introduction

The goal of this assignment is that your should practice representing data using tuples and structs and, understand how a program can be divided into modules to implement abstract data structures.

This is also an exercise in how to model a complex system and refresh your knowledge of vector arithmetic. Hopefully it's also quite fun to see how you can create your own images of a three dimensional world.

1 A ray tracer

In order to explain a ray tracer in a three dimensional world, we will describe all necessary steps using a two dimensional model. Since we will describe everything using vector arithmetic it can easily be extended to a three dimensional model. Images generated in a two dimensional world will of course not be very thrilling but we will be able to describe the necessary steps.

In Fig. 1 we see a sphere in a two dimensional Cartesian coordinate system. A *camera* is also positioned in the room consisting of a *canvas* and an *eye*. When generating an image we need to trace as many rays as possible starting in the eye of the camera and passing through the *pixels* of the canvas. If we can determine that a ray intersect with an object we can color the pixel thus generating an image on the canvas.

The beauty of vector arithmetic is that if we understand how to do the necessary calculations in a two dimensional space then we also know how

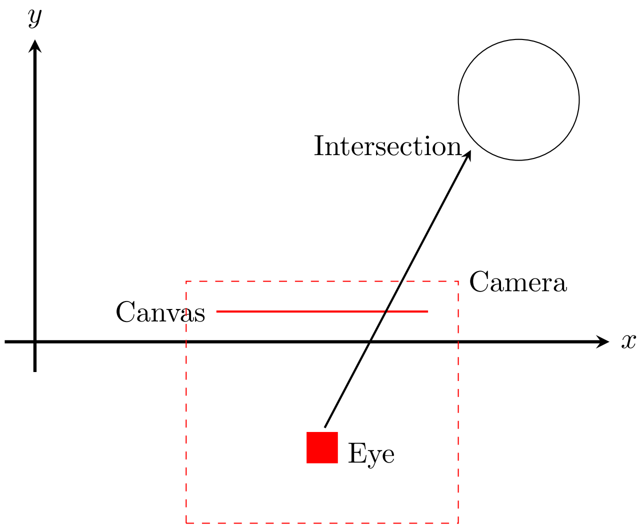

Figure 1: A model of the world.

to do it in a three dimensional space. If we extend this model to a three dimensional world, we would have a sphere instead of a circle and the canvas would be a rectangular plane. The eye would still be a point in this space and we would track rays from the eye through each *x-y pixel* of the canvas.

To model our world and do the necessary computations we need to solve the following problems:

- represent and do calculations on vectors and rays
- represent objects such as spheres
- determine if a ray intersects an object
- represent a camera i.e. the location of the eye and the canvas

When we solve these problems we will try to separate them as much as possible form each other. We will divide our system into modules were each module is responsible for one sub-problem or abstraction. If we do it right the main program can describe operations in a much higher level; instead of talking about x, y and z coordinates it can talk about rays, objects and intersections.

2 Vectors and rays

The first task is to create a module that will handle all vector operations. We will need to represent vectors and be able to do the following operations.

- $a\vec{x}$: scalar multiplication
- $\vec{x} - \vec{y}$: subtraction
- $\vec{x} + \vec{y}$: addition
- $\|\vec{x}\|$: norm, or length, of a vector
- $\vec{x} \cdot \vec{y}$: scalar product (dot product)
- $|\vec{x}|$: normalized vector

If we restrict the system to only work with three dimensional vectors we have a natural way of representation: a tuple with three elements, the x, y and z components i.e. {x, y, z}.

Create a new file vector.ex and declare a new module with the following exported functions.

```
defmodule Vector do

  def smul({x1, x2, x3}, s) do ... end

  def sub({x1, x2, x3}, {y1, y2, y3}) do ... end

  def add({x1, x2, x3}, {y1, y2, y3}) do ... end

  def dot({x1, x2, x3}, {y1, y2, y3}) do ... end

  def normalize(x) do ... end

  def norm({x1, x2, x3}) do ... end

end
```

Vector arithmetic

The first functions, scalar multiplication, addition and subtraction should be quite easy to implement.

$$\langle x_1, x_2, x_3 \rangle s = \langle x_1 s, x_2 s, x_3 s \rangle$$

$$\langle x_1, x_2, x_3 \rangle + \langle y_1, y_2, y_3 \rangle = \langle x_1 + y_1, x_x + y_2, x_3 + y_3 \rangle$$

To implement the norm, dot product and normalization of a vector you might have to go through your book in linear algebra but you should have it up and running quite quickly.

$$\|\vec{x}\| = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

$$\vec{x} \cdot \vec{y} = \langle x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3 \rangle$$

$$|\vec{x}| = \vec{x}/\|\vec{x}\|$$

In this implementation we actually expose the representation of a vector i.e. the users of this module will know that vectors are represented by tuples with three elements. This is not the best solution but unfortunately a very convenient solution.

Rays

We will however create a more proper data structure to represent a *ray*. A ray has an origin, represented by a vector, and a direction, represented by a normalized vector. When we implement rays we will use Elixir structs. A struct is always associated with a module and gives us a convenient way of referencing elements of a data structure. Instead of having to remember which element of a tuple a particular element is we can refer to them by name.

Create a module `Ray` and define the following struct. We provide a default position `{0,0,0}` and direction `{1,1,1}`. If none is given when we create a ray the default values will be used.

```
defmodule Ray do

  defstruct pos: {0, 0, 0}, dir: {1, 1, 1}
```

```
end
```

We now have the mathematical tools to describe a world of obkects.

3 Objects

Our next task is to create data structures that can represent the objects in our world. We will keep things simple and only handle spheres. We should however implement the spheres in a way that makes it easy to add new objects in the future.

We will use Elixir *protocols* to define a common interface that all objects should implement. Similarly to defining a module we define a protocol called `Object`. The protocol will so far only specify one function `intersect/2` that will determine if a ray intersects an object.

```
defprotocol Object do

  def intersect(object, ray)

end
```

As we define different object we will instantiate the `Object` protocol by defining the `intersect/2` function.

Spheres

When we decide on the representation we need to think about the operations we should perform; a representation is efficient if it allows the operations to be efficiently implemented. When we are talking about rays and spheres, we might not have much choice but it's important to start thinking about the operations that should be performed.

The operations that we will perform over and over again is to determine if a ray intersects with another object such as a sphere. If we model our objects in a Cartesian space we will be able to determine intersections quite easily.

A sphere will have an position and a radius. The position is represented by a vector and the radius by a number. We create a struct with two properties and default values.

```
defmodule Sphere do

  defstruct pos: {0, 0, 0}, radius: 2

end
```

Intersection

The tricky part is of course to determine if a ray will intersect a sphere but this is actually easily determined if we remember our linear algebra.

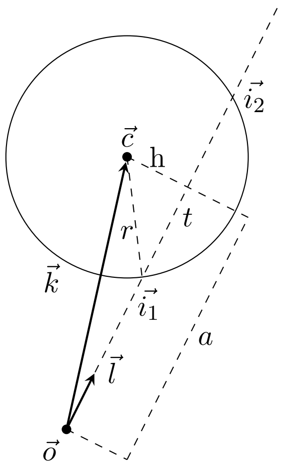

Figure 2: Intersection of ray and sphere.

In Fig. 2 we see a ray intersecting a circle. We want to find the intersection points $\vec{i_1}$ and $\vec{i_1}$.We can do this by first calculate the length a and this is done by taking the dot product of $\vec{k}$ and $\vec{l}$. The dot product will project the vector $\vec{k}$ on $\vec{l}$ thus giving us the length a. The vector $\vec{k}$ is of course easily calculated since we know the origin of the ray $\vec{o}$ and the center of the circle $\vec{c}$.

Note that we here talk about the circle while our real model would contain a sphere - this is fine, the operations are the same.

The length of the vector $\vec{k}$ is of course $\|\vec{k}\|$ and if we know this we can calculate h using Pythagoras' theorem. Since we know that the radius of the sphere is r we can again rely on Pythagoras and calculate t^2.

$$t^2 = r^2 - h^2$$

If it turns out that t^2 is a negative value, it means that the ray does not intersect the sphere. This is our criteria for answering if we intersect the object or not. If t^2 is positive we calculate t and then of course obtain two alternatives t and $-t$. We now calculate two distances $d_1 = a - t$ and $d_2 = a + t$. This is the distance to the points of intersections from the origin of the ray $\vec{o}$. If either value is negative it means that the intersection point is behind us; if only one value is negative we are actually inside the sphere. If both values are positive we return the smallest value since this is the surface that we will actually see.

Object protocol

Implement the function `intersect/2` that checks if ray intersects a sphere; return `:no` if it does not and `{:ok, d}`, where `d` is the closest distance, if it does. We implement the function in the `Sphere` module but use the `defimpl` construct to declare this as the `Object` protocol implementation.

```
defimpl Object do

  def intersect(sphere = %Sphere{}, ray = %Ray{}) do
    :
    :
  end
end
```

The function `intersect/2` is now reached using the call `Object.intersect/2` since it is defined as an `Object` protocol.

4 The camera

We have now done half of the job, you will soon create your first image but we first need to represent the camera. It turns out that we have a lot of options when defining what the camera looks like; we of course need to define where in the room it is and where it is pointing but also what kind of lens it has. This is probably the most complicated part of the implementation but we will give it a try.

The name of the game

In the end we would like to have a representation of a camera that will allow us to ask for a ray that starts in the focal point (or origin) of the camera and runs through a given $\langle x, y\rangle$ coordinate of the canvas. If we know that the canvas is of size 800×600 then we can ask for the ray that runs through $\langle 230, 170\rangle$ and be given a a ray. This ray will then be compared to all the objects in the world and the closest intersection point will determine the color of the $\langle 230, 170\rangle$ pixel.

When you think about representation, then always think about what you're going to use the object for. This will allow you to represent the object in a way that makes the operations easier to perform. Also think about how you would like to talk about the object, the easiest way to describe an object might not be the best way to represent it.

Properties of a camera

If you have not used a large camera you might not have thought about how different lenses changes the picture but think about the difference between a "fish-eye" and telephoto lens. The difference has to to with the *focal length*, the length from the lens to the focal point; the important factor is the ratio between the width of the *film* and the focal length. When using a 35mm film a focal length of 50mm gave a "normal" lens i.e. a lens that gave images that looked normal.

It is thus important that we can describe a *canvas*: its size, orientation and position in relation to the *origin* of the camera. In Fig. 3 we see the elements that we need to represent: the origin described by a vector $\vec{o}$, a vector $\vec{f}$ that give us the direction and distance to the center of the canvas and two vectors that give us the vertical, $\vec{v}$, and horizontal, $\vec{h}$, direction of the canvas.

We might take for granted that the plane of the canvas is orthogonal to the direction; this is not strictly necessary but if it is not, we will have very strange projections of the image (a technique that is actually used and if you want to know more you can search for the "Scheimpflug principle").

Note that the vertical and horizontal orientation are represented as two vectors. This will allow us to create a ray from the origin through any coordinate of the canvas.

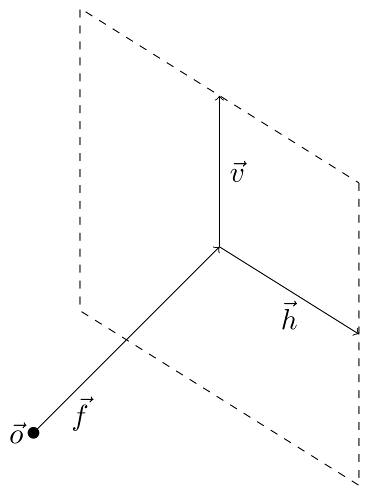

Figure 3: Representing a camera.

A camera with a normal lens, positioned at $\langle 0, 0, 0\rangle$ and pointing straight into the picture, could be described as follows:

- position: $\langle 0, 0, 0\rangle$
- direction: $\langle 0, 0, 1200\rangle$
- horizontal: $\langle 960, 0, 0\rangle$
- vertical: $\langle 0, 540, 0\rangle$

This would give us a canvas of size 1920×1080 at a distance from the origin of 1600 (which will approximately give us a "normal" lens).

To minimize the computation needed when calculating the rays we could represent the camera by a position and a vector to the upper left corner of the canvas $\vec{c}$. If we then have two vectors that represent the distance between pixels moving to the right $\vec{r}$ and moving down $\vec{d}$, we can easily calculate the normalized vector to any pixel in the canvas.

$$pixel(x, y) = |\vec{c} + x * \vec{r} + y * \vec{d}|$$

We therefore represent the camera by position, direction to the upper left corner and the two vectors that describes the distance to the first pixel to the right and the first pixel down. Why the upper corner, why move down, why not lower left corner? Turns out that when we talk about images we often count the rows going down so this will makes things easier. We also keep the size of the canvas so we know which rays that we should produce.

Open up a module `Camera` and defined the following struct.

```
defmodule Camera do

  defstruct pos: nil, corner: nil, right: nil, down: nil, size: nil

end
```

It will also be very handy to have a function that returns a default camera so that we don't have to think about the different parameters. This default camera can be positions at $\langle 0, 0, 0 \rangle$ and point straight forward (in the z direction). We can give it a parameter that is the size of the image that we want to generate; you will have to calculate the rest of the parameters.

```
def normal(size) do
  {width, height} = size
  d = width * 1.2
  h = width / 2
  v = height / 2
  corner = ...
    :
    :
  %Camera{pos: {0, 0, 0}, corner: corner, ..., ..., size: size)
end
```

Given a camera we now need to calculate a ray that passes through a given coordinate or pixel. This should be straight forward given that we created our representation with this in mind.

```
def ray(camera, x, y) do
  origin = ...   # the origin of the ray
  x_pos = ...    # a vector from the corner to the x column
  y_pos = ...    # a vector from the corner to the y row
  pixle = ...    # a vector from origin to the pixle
  dir = ...      # the normalized vector
  %Ray{pos: origin, dir: dir}
end
```

The world

The world is simply a list of objects. We will extend the world to also hold other things but for our first test this will be sufficient.

```
defmodule World do

  defstruct objects: []

end
```

That completes all the modules that we need to represent the world, high time to generate some images.

5 The tracer

Open a module called `Tracer`; this will be the main module where the images are created. We will define a function that takes a camera and a description of the world and returns an image.

An image will be represented by a list of rows where each row is a list of *rgb values*. The rgb values are tuples of three elements where each element is a floating point value between 0 and 1. A green-blue color could thus be represented by the tuple {`0, 0.8, 0.3`} We will later use a procedure to print this image to a file that you hopefully can open in a viewer of your choice.

We will start slowly and render a black and white image. This will not impress anyone but your mother, but it will be a start that we then will extend quite easily. We will build the tracer *bottom up* which will give us the opportunity to test things as we implement it.

Intersect

The first thing we will do is to determine which object a ray intersects, if any, from a list of objects. We know that we can call `intersect/2` using the `Objects` protocol but now we have a list of objects and we want to find the closest point of intersection.

The `intersect/2` function returns either `{:ok, d}` or `:no` so it should be an easy task to find the object with the closest point of intersection. We will here use the higher order construct `List.foldl/3` but you could implement it by hand in just as many lines.

```
def intersect(ray, objects) do
  List.foldl(objects, {:inf, nil}, fn(object, sofar) ->
    {dist, _} = sofar

    case Object.intersect(object, ray) do
      {:ok, d} when d < dist ->
        {d, object}
      _ ->
        sofar
    end
  end)
end
```

The function `intersect/2` will tell us if a ray intersects an object and it will also tell us the distance to this object. If the ray does not intersect an object it will return `{:inf, nil}` (using a value like this is called a sentinel, a value that we know will be higher than any other value).

The use of the `Object` protocol allows us to have objects of different kinds. We only have spheres now but can add any objects as long as they implement the `Object` protocol.

Tracing

We now define a two functions: `trace/4` and `trace/2`. The latter will use `intersect/2`, check the outcome and then return either a black or white rgb-value depending on if we intersected an object. The first will use the latter but here we give the x and y values of the pixel that we are looking for.

```
def trace(x, y, camera, objects) do
  ray = ...
  trace(ray, objects)
end

def trace(ray, objects) do
```

```
  case intersect(ray, objects) do
    ... ->
      @black
    ... ->
      @white
  end
end
```

Here we use Elixir module attributes to define the rgb-values for black and white. You will have to add these in the beginning of the module.

```
  @black {0, 0, 0}
  @white {1, 1, 1}
```

The only thing that is left is to trace every possible ray of the camera; this is neatly done using list comprehension. The function `tracer/2` will return a list of rows where each row is a list of rgb values that describes the image.

```
def tracer(camera, objects) do
  {w, h} = camera.size
  xs = Enum.to_list(1..w)
  ys = Enum.to_list(1..h)
  for y <- ys, do: for(x <- xs, do: trace(x, y, camera, objects))
end
```

We of course want to look at the image we have created and to do this we somehow have to convert it into a image file. There are many image formats to choose from but most are compressed and it is not trivial to write a jpeg encoder. The file format that we will use is very inefficient but it is very easy to generate a file. In the appendix App A you will find a module that will take an image, as we generates it, and writes a *ppm* file. Not all image viewers can open a ppm file so you might have to try several before finding one that works.

It is now quite easy to generate an image, all we have to do is to describe the world and then take a snap shot. Create a module called `Test` and describe your first image.

```
defmodule Test do

  def snap do
    camera = Camera.normal({800, 600})

    obj1 = %Sphere{pos: 140, pos: {0, 0, 700})
    obj2 = %Sphere{radius: 50, pos: {200, 0, 600}}
    obj3 = %Sphere{radius: 50, pos: {-80, 0, 400}}

    image = Tracer.tracer(camera, [obj1, obj2, obj3])
    PPM.write("test.ppm", image)
  end

end
```

If you look at your picture you will hopefully see three white circles on a black background. One circle (the image of `obj3` in the description above) is closer to the camera and is partly blocking the larger circle (that is the image of `obj1`). This is not very existing but you will see that it is very easy to extend this simple ray tracer.

6 Extensions

We will do three extensions to the tracer: colors, lights and reflections. The first extension is quite simple while the last one requires that you repeat you knowledge i geometry.

Adding colors

Turning the image into a color image is two simple changes. First of all we extend the `Sphere` struct structure to also include a color element. We can have a default color if we want, so that all spheres have colors.

```
@color {1.0, 0.4, 0.4}

defstruct pos: {0, 0, 0}, radius: 2 , color: @color
```

Once we have spheres with colors, we simply change the `trace/3` function so that it will return the color of the object rather than the default white.

Describe a new snap-shot where you have some colorful spheres and see what the image looks like.

6.1 Lights

Extend the `World` module and describe a world struct to also hold a list of lights and a background color.

```
@background {0, 0, 0}

defstruct objects: [], lights: [], background: @background
```

The question is now what a light source looks like; we create a new module `Light` that will handle all aspects of it. The lights themselves are simple to model since we only give them a position and a color.

```
defmodule Light do

  defstruct pos: nil, color: {1.0, 1.0, 1.0}

end
```

The question now is how we are going to use the lights; we could probably have a whole course on how light sources are combined in a ray tracer but we will try to keep it simple.

Let's look at the trace function, it detects if a ray hits an object and then returns the object and the distance to this object. Since we know the direction of the ray we can easily describe the point in space where the ray hits the object.

What if we ask if a ray starting in this point and in the direction of a light source, hits any object in the world. If it does not intersect with any object it should be illuminated by the light source. If we examine all light sources we could determine which sources that illuminates the point. Combined with the color of the object we can determine the color of the corresponding pixel. Note that you have all the pieces of this puzzle, it's just a matter of combining light sources and the color of the object.

```
defp trace(ray, world) do
  objects = world.objects

  case intersect(ray, objects) do
    {:inf, _} ->
      world.background
    {dist, obj} ->
      pos = ray.pos
      dir = ray.direction
      point = Vector.add(pos, Vector.smul(dir, dist - @delta))
      normal = Sphere.normal(point, obj)
      visible = visible(point, world.lights, objects)
      illumination = Light.combine(point, normal, visible)
      Light.illuminate(obj, illumination, world)
  end
end
```

In the code above there are two things that needs some explanation: the `@delta` and the use of the *normal vector*. The delta is a hack that we need to do since floating points are not exact, or rather since a point that is actually "in" the surface of an object might be shadowed by the object itself. By raising the point a small distance from the surface we avoid being lured into thinking that we are in the surface or even worse below the surface; the delta that I use is 0.001 and it works fine.

The second thing is the normal vector that we use when combining the light sources. A light that hits the surface at an angle will contribute to less illumination compared to a light that hits it straight from above. You can first try to combine the light sources without taking this into effect but you will see that the light is very sharp, either it illuminates the surface or it does not. Using the normal vector does require that you do some more vector arithmetic but it turns out to be quite simple.

The normal vector $\vec{n}$ is easily calculate since we know the point of intersection $\vec{i}$ and the center of the sphere $\vec{c}$. If we have other objects we would of course have to do something else.

$$\vec{n} = |\vec{i} - \vec{c}|$$

The contribution a of a light source at $\vec{s}$ to the point $\vec{i}$ on a surface with normal vector $\vec{n}$ is:

$$a = |\vec{s} - \vec{i}| \cdot \vec{n}$$

What we are doing here is to first calculate the vector from $\vec{i}$ to $\vec{s}$ and then normalize this. Then we do the dot product with the normal vector to obtain a number between 0 and 1. A light source that is orthogonal to the normal vector will not contribute at all while a light source in exactly the same direction will contribute with its full strength.

All the light sources can be added together but we of course need to do the addition in a special way. When you add two probabilities p and q then you would write:

$$1 - ((1 - p) \times (1 - q))$$

And we should do the same thing here (do some thinking). If you get it right your images will get a lot more live and will start to look like something that you could show to someone besides your mother.

Reflections

The third extension that we will look at is reflections; sounds tricky but it turns out to be even simpler than adding lights. What we wan to do is to calculate a reflecting ray in the point of intersection and then calculate the contribution from this angle. The contribution can of course be calculated using a recursive step since this is exactly what the tracer module will do; given a origin and a normal vector calculate what is visible in this direction.

So if we do a recursive step and find that the reflection has a particular color then we need to ask ourselves how much of this color should be added to the point of intersection. This is where you will start to think about *brilliance* i.e. how much the surface acts as a mirror. Again, you could spend the rest of the week thinking about how a metal surface is different from wood but we might also just describe the level of reflection by a number from 0 to 1.

Add a property to the sphere object that gives us the brilliance and then use this value when you calculate how much the reflection should be visible. You then modify the `tracer/2` function so it takes another argument which will be the depth, or number of reflections. If the depth is zero we're done and simply return the background color, otherwise we calculate the intersection point, calculate the contribution from light sources and then also add the reflection that you obtain from a recursive call to the function (remember to decrement the depth value).

Once we have the brilliance we might want to change the way we add light sources since a light source that is in the direction of the reflection would be visible in a sphere with high brilliance. If you start to dig into this you will find that you have enough to do for a day or two.

And more

Another thing that we might want to explore is of course to add more objects. A plane would be fairly easy to describe and the intersection can be found if you do some reading.

We could also start to add texture or images to the surface of objects. An image could of course be mapped to a rectangular plane or cylinder and you would see the distorted photo in your rendered image.

A tricky thing to add is transparency; an object can be made to look like colored glass. To do this you would simply calculate two contributing rays at a point of intersection. One is in the direction of the reflection but the other one is continuing through the object, this is called the *refraction.* The refraction is of course, if you remember what you learned in secondary school, slightly bent depending on the index of the material. You would thus use the refraction index to calculate the direction of the ray and then take the transparency into account when you add the contribution of the refraction.

This becomes tricky since you would have to keep track of if we enter of exit an object, the *current* refraction index and the index of the material outside of an object. If we only consider air and solid spheres that do not overlap it's easier but the general case becomes problematic.

7 Summary

This tutorial was not about ray tracing but about how you can work with tuples, structs and modules to build a larger program where try to hide the internal data representation as much as possible.

If you have programmed in for example Java, which is a statically typed object orientated language, you have seen better ways of doing this. The dynamically typed languages have for good and worse less support to achieve this; the struct construct in Elixir is a addition to the language that gives you some support but it's only half way.

If your familiar with object orientated languages you have probably thought about describing out objects in a hierarchical way; all object have a position and color, spheres are object that have a radius etc. This would definitely make our life easier but again, the dynamic nature of Elixir makes it harder to provide.

When looking at you program you could see the different modules as the building blocks of abstractions. If you have done it right only the vector module knows how vectors are handled (all though we cheated and anyone that creates an object must also know how a vector is represented. Anything that is dealing with rays, objects and intersections should be in the objects module and everything that handles colors and lights should be in the light module. Dividing you program up in modules gives you a similar tool for abstractions as the classes in Java (not exactly by similar). Learning how to divide a program into different modules of abstraction is maybe the most important skill of a good programmer.

Even if the tutorial was not about ray tracing, I hope that you have generated some nice images and have a better understanding of what your graphic card is doing when you spawn in BF.

A Module ppm.ex

```
defmodule PPM do

  # write(Name, Image) The image is a list of rows, each row a list of
  # tuples {R,G,B} where the values are floats from 0 to 1. The image
  # is written using PPM format P6 and color depth 0-255. This means that
  # each tuple is written as three bytes.

  def write(name, image) do
    height = length(image)
    width = length(List.first(image))
    {:ok, fd} = File.open(name, [:write])
    IO.puts(fd, "P6")
    IO.puts(fd, "#generated by ppm.ex")
    IO.puts(fd, "#{width} #{height}")
    IO.puts(fd, "255")
    rows(image, fd)
    File.close(fd)
  end

  defp rows(rows, fd) do
    Enum.each(rows, fn r ->
      colors = row(r)
      IO.write(fd, colors)
    end)
  end

  defp row(row) do
    List.foldr(row, [], fn({r, g, b}, a) ->
      [trunc(r * 255), trunc(g * 255), trunc(b * 255) | a]
    end)
  end

end
```