To show how to work with some Elixir programming constructs and to discuss representation and modeling, we will implement a small ray tracer.

Architecture

- **vector**: vector arithmetic
- **ray**: the description of a ray
- **sphere**: a sphere object
- **object**: a protocol for all objects
- **camera**: the camera position, direction and characteristics
- **tracer**: responsible for the tracing of rays
- **ppm**: how to generate a .ppm file

and possibly some more

The basic idea of ray tracing:
We first need a module to handle vector arithmetic:

- Do we need to handle vectors of arbitrary dimensions?
- How do we represent vectors?
- What basic operations should we implement?

- $a\vec{x}$: 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 $\|\vec{x}\| = \vec{x}/\|\vec{x}\|$

The notation for a normalized vector differ, sometimes it is written as $\hat{x}$

---

defmodule Vector do

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

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

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

def norm({x1,x2,x3}) do
  :math:sqrt(x1*x1 + x2*x2 + x3*x3)
end

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

def normalize(x) do
  n = norm(x)
  smul(x, 1 / n)
end

---

We now define how to represent object and rays.

- **ray**: position and direction
- **sphere**: position, radius, ...
- **object**: a protocol for all objects
A ray is defined by a position and a direction. The position is a vector (a place in the space) and the direction is a unit vector.

```
defmodule Ray do
  require Record
  Record.defrecord(:ray, pos: {0, 0, 0}, dir: {0, 0, 1})
end
```

A number of macros defined to help us handle properties of rays.

```
foo = Ray.ray(pos: {3, 2, 4}, dir: {0.447, 0.894, 0})
Ray.ray(pos: pos) = foo
p = Ray.ray(foo, :pos)
```

Elxir protocols

All objects in the world should provide a function that can determine if it intersects with a ray.

```
defprotocol Object do
  def intersect(object, ray)
end
```

Each object will implement the function intersect/2.

A sphere is defined by:

```
defmodule Sphere do
  require Ray
  defstruct pos: {0, 0, 0}, radius: 2
end
```

More properties will be added later.
Structs are named maps. Gives us a nice syntax to handle the objects.

```elixir
template = %Sphere{pos: {3, 2, 4}, dir: {0.447, 0.894, 0}}
```

```elixir
p = foo.pos
```

Note, access is \( \log(n) \) of number of properties, not as efficient as tuples nor records.

---

### intersection

\[
\vec{k} = \vec{c} - \vec{o} \\
a = \vec{l} \cdot \vec{k} \\
\|\vec{k}\|^2 = a^2 + h^2 \\
r^2 = h^2 + t^2 \\
t^2 = a^2 - \|\vec{k}\|^2 + r^2 \\
\vec{i} = \vec{o} + d\vec{l} \\
d_i = a \pm t \\
\text{if } d_i < 0 \text{ then } \vec{i} \text{ is behind the origin } \vec{o}
\]

```elixir
defimpl Object do
def intersect(sphere, ray) do
  Ray.ray(ray, pos: o, dir: l)
  k = Vector.sub(sphere.pos, o)
  a = Vector.dot(l, k)
  a2 = :math.pow(a, 2)
  k2 = :math.pow(Vector.norm(k), 2)
  r2 = :math.pow(sphere.radius, 2)
  t2 = a2 - k2 + r2
  closest(t2, a)
  
  :end
end
```
What properties do we have?

- position: in space
- direction: a unit vector
- size of picture: width and height
- focal length: distance to canvas
- resolution: pixels per distance

defmodule Camera do
  require Record

  Record.defrecord(:camera, pos: nil, corner: nil, right: nil, down: nil, size: nil)
end
a normal lens pointing forward

```ruby
def normal(size) do
  {width, height} = size
  d = width * 1.2
  h = width / 2
  v = height / 2
  corner = {-h, v, d}
  pos = {0, 0, 0}
  right = {1, 0, 0}
  down = {0, -1, 0}
  Camera.camera(pos: pos, corner: corner, .... )
end
```

we have everything

```
defmodule Tracer do
  @black {0, 0, 0}
  @white {1, 1, 1}
  def tracer(camera, objects) do
    {w, h} = Camera.camera(camera, :size)
    for y <- 1..h, do: for(x <- 1..w, do: trace(x, y, camera, objects))
  end
  def trace(x, y, camera, objects) do
    ray = Camera.ray(camera, x, y)
    trace(ray, objects)
  end
end
```

def ray(camera, col, row) do
  Camera.camera(camera,
  pos: o, right: r, down: d,
  corner: c) = camera
  x = Vector.smul(r, col)
  y = Vector.smul(d, row)
  v = Vector.add(x, y)
  p = Vector.add(c, v)
  d = Vector.normalize(p)
  Ray.ray(pos: o, dir: d)
end

the tracer

```
Given a camera we want to find the rays from the camera “origin” to the \( \{ \text{col}, \text{row} \} \) position of the canvas.
```

```
def ray(camera, col, row) do
  Camera.camera(camera,
  pos: o, right: r, down: d,
  corner: c) = camera
  x = Vector.smul(r, col)
  y = Vector.smul(d, row)
  v = Vector.add(x, y)
  p = Vector.add(c, v)
  d = Vector.normalize(p)
  Ray.ray(pos: o, dir: d)
end
```

```
defmodule Tracer do
  @black {0, 0, 0}
  @white {1, 1, 1}
  def tracer(camera, objects) do
    {w, h} = Camera.camera(camera, :size)
    for y <- 1..h, do: for(x <- 1..w, do: trace(x, y, camera, objects))
  end
  def trace(x, y, camera, objects) do
    ray = Camera.ray(camera, x, y)
    trace(ray, objects)
  end
end
```

```
Given a camera we want to find the rays from the camera “origin” to the \( \{ \text{col}, \text{row} \} \) position of the canvas.
```

Given a camera we want to find the rays from the camera “origin” to the \( \{ \text{col}, \text{row} \} \) position of the canvas.
tracing a ray

```elixir
def trace(ray, objects) do
  case intersect(ray, objects) do
    {:inf, _} ->
      @black
    {_, _} ->
      @white
  end
end
```

the last piece

```elixir
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
end
```

time to test

```elixir
defmodule Snap do
  def snap(0) do
    camera = Camera.normal({800, 600})
    obj1 = %Sphere{radius: 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("snap0.ppm", image)
    end
  end
end
```

snap0.ppm

![Snap0.png](attachment:Snap0.png)
Let's add some colors to the spheres.

defprotocol Object do
  def intersect(object, ray)
  def color(object)
end

@color {1.0, 0.4, 0.4}

defstruct radius: 2, pos: {0, 0, 0}, color: @color
defimpl Object do
  def color(sphere) do
    sphere.color
  end
end

def trace(ray, objects) do
  case intersect(ray, objects) do
    {:inf, _} -> @black
    {_, object} -> Object.color(object)
  end
end

We want to add some lights to the world.
Lights have a position and a color.
The color of an intersection point is determined by the color of the object combined with the colors from the lights.

- **lights**: handles everything that has to do with lights and colors.

the representation of colors is a RGB tuple of floats 0..1.0 i.e. \{1.0, 0.5, 0.2\}
The normal vector \( \vec{n} \) is the normal unit vector, i.e., perpendicular to the sphere, at the point of intersection.

\[ \vec{n} = |\vec{i} - \vec{c}| \]

Will come in handy when we calculate reflection and illumination.

defmodule World do
  @background {0, 0, 0}
  @ambient {0.3, 0.3, 0.3}
  require Record
  Record.defrecord(:world, objects: [], lights: [], background: @background, ambient: @ambient)
end

A more convenient way to handle lack of globally accessible data structures.

Calculating the color

Find all visible lights from the point of intersection; combine the lights give the normal vector and illuminate the surface.

In the tracer, when we have found an intersecting object:

```Clojure
  case intersect(ray, objects) do
    {:inf, _} ->
      @black
    {d, obj} ->
      Ray.ray(pos: o, dir: l) = ray
      i = Vector.add(o, Vector.smul(l, d - @delta))
      normal = Object.normal(obj, i)
      visible = visible(i, World.world(world, :lights), objects)
      illumination = Light.combine(i, normal, visible)
      Light.illuminate(obj, illumination, world)
```

```
; snap2.ppm
```
The color of an intersection point depends on:
- color of the object
- combination of light sources
- reflection from other objects

```
defp trace(_ray, 0, world) do
  World.world(world, :background)
end

defp trace(ray, depth, world) do
  objects = World.world(world, :objects)
  :
  :
  reflection = trace(r, depth - 1, world)
  Light.illuminate(obj, reflection, illumination, world)
end
end
```

This was only scratching the surface of ray tracing.
from an architecture point of view

- divide program into areas of responsibility
- think about abstractions
- modules are similar to class definitions
- a static type system would have helped us (structa are only halfway)