A Ray Tracer

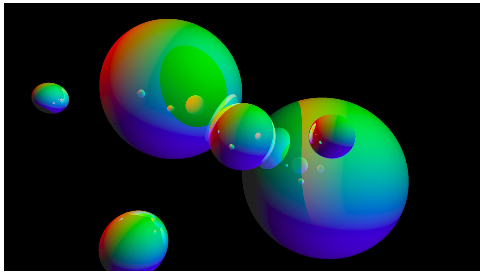
Johan Montelius

KTH

VT24

A programming example

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



Architecture

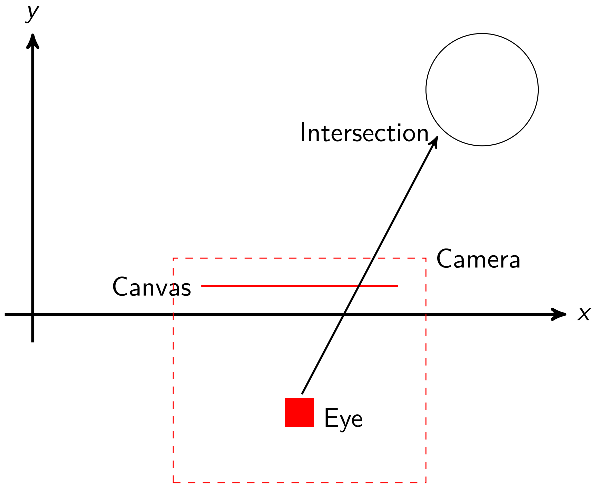
modules that we will implement

- **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

ray tracing

The basic idea of ray tracing:



vector arithmetic

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?

vector arithmetic

- $a\vec{x}$: 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)
- $\hat{x}$: normalized vector $\hat{x} = \vec{x}/\|\vec{x}\|$

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

vector arithmetic

```
defmodule Vector do

def smul({x,y,z}, s) do
  {x*s, y*s, z*s}
end

def add({x1,y1,z1}, {x2,y2,z2}) do
  {x1+x2, y1+y2, z1+z2}
end

def sub({x1,y1,z1}, {x2,y2,z2}) do
  {x1-x2, y1-y2, z1-z2}
end

def norm({x,y,z}) do
  :math.sqrt(x*x + y*y + z*z)
end

def dot({x1,y1,z1}, {x2,y2,z2}) do
  x1*x2 + y1*y2 + z1*z2
end

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


polymorphism

polymorphism : the quality or state of existing in or assuming different forms

```
  ... x = {1, 3, 2}; y = {3, 2, 4};   x + y

def add({x1,x2,x3}, {y1,y2,y3}) do
   {x1+y1, x2+y2, x3+y3}
end
def add( x, y ) when is_number(x) and is_number(y) do x + y end

def add( :sprit, :fyrverkeri ) do :intebra end
```

Plolymorphism is more efficient and easier to support in a statically typed language.

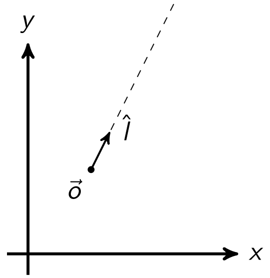
objects

We now define how to represent object and rays.

- **ray:** position and direction
- **sphere:** position, radius, ...
- **object:** a *protocol* for all obejcts

rays

A ray is defined by an position and a direction. The position is a vector (a place in the space) and the direction is a *unit vector*.




```
defmodule Ray do

  defstruct( pos: {0, 0, 0}, dir: {0, 0, 1})
end
```


tuples and structs

- a vector: `{2, 3, 1}`
- a ray:

```
p = ray.pos   d = ray.dir
```

```
%Ray{pos: p, dir: d}
```

Note, access is lg(n) of number of properties, not as efficent as tuples.

Elxir protocols

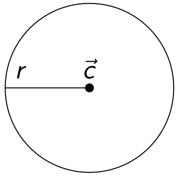
All objects in the world should provide a function that can determine if it intersects with a ray.
Introducing protocols:

```
defprotocol Object do

  def intersect(object, ray)

end
```

Each object will implement the function intersect/2.

spheres

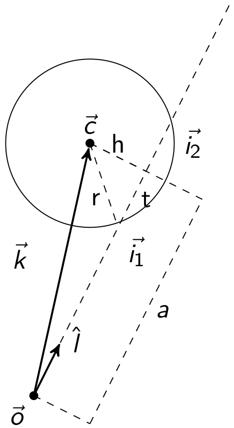
A sphere is defined by:




```
defmodule Sphere do

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

end
```

more properties will be added later

intersection



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

intersection

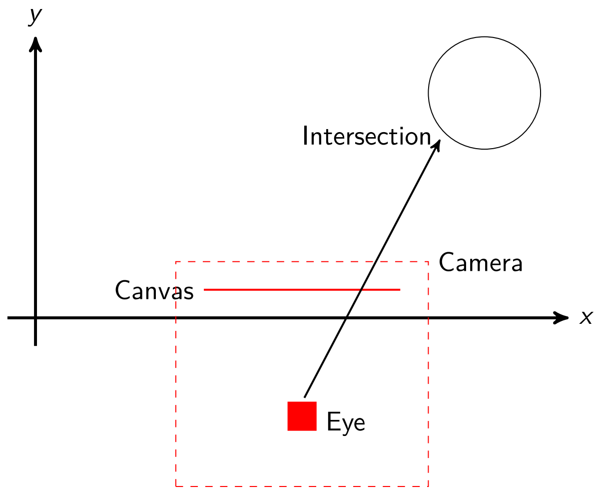
```
defimpl Object do

 def intersect(sphere, ray) do
   k = Vector.sub(sphere.pos, ray.pos)
   a = Vector.dot(ray.dir, 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
```

$$\vec{k} = \vec{c} - \vec{o}$$

$$a = \hat{l} \cdot \vec{k}$$

$$t^2 = a^2 - \|k\|^2 + r^2$$

ok, what else?



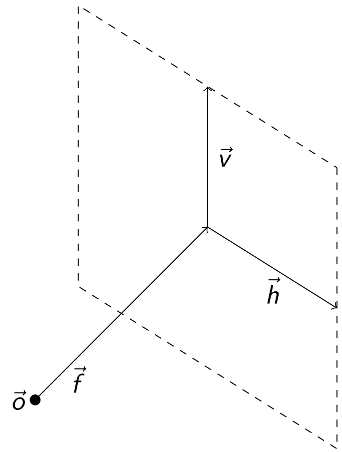
the camera

What properties do we have?

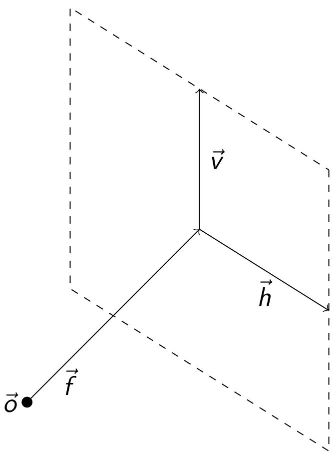
- position : in space
- direction : a unit vector
- size of picture : width and height
- focal length : distance to canvas
- resolution: pixles per distance

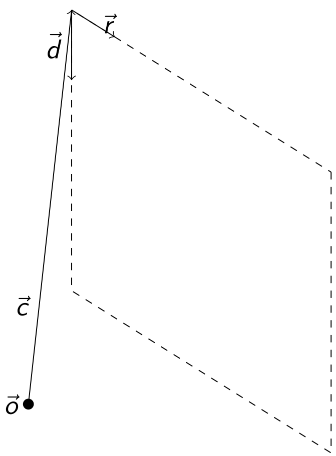
the camera

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



the camera





a simple camera

```
defmodule Camera do

  defstruct( pos: nil, corner: nil,
             right: nil, down: nil, size: nil)
```

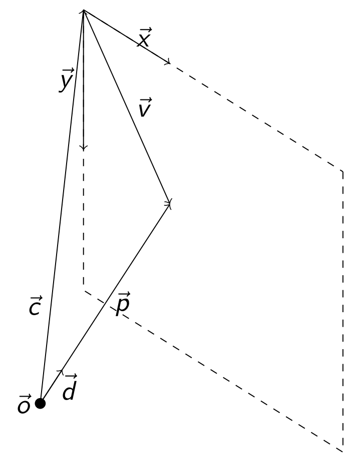
a normal lens pointing forward

```
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{pos: pos, corner: corner, .... }
end
```

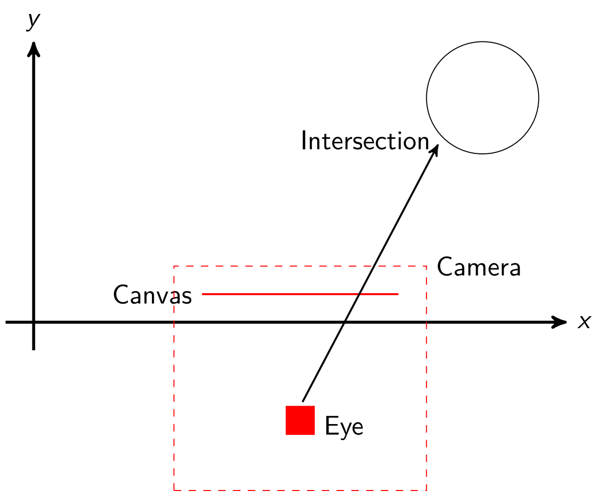

rays

Given a camera we want to find the rays from the camera "origin" to the {col,row} position of the canvas.

```
def ray(camera, col, row) do
  x = Vector.smul(camera.right, col)
  y = Vector.smul(camera.down, row)
  v = Vector.add(x, y)
  p = Vector.add(camera.corner, v)
  dir = Vector.normalize(p)
  %Ray{pos: camera.pos, dir: dir}
end
```



we have everything



the tracer

```
defmodule Tracer do

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


  def tracer(camera, objects) do
    {w, h} = 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
```

tracing a ray

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

    {_, _} ->
      @white
  end
```

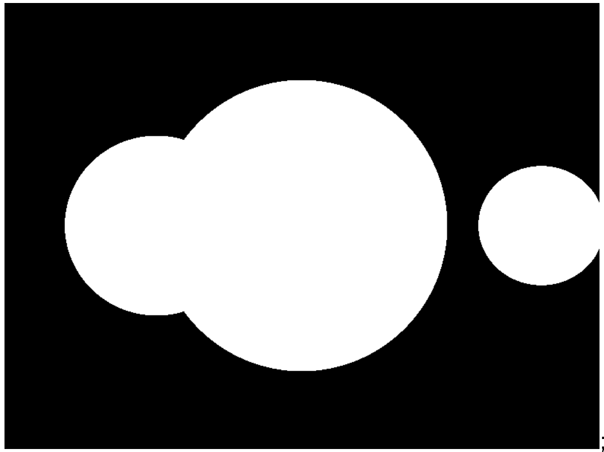

the last piece

```
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
```


time to test

```
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
```


snap0.ppm



colors

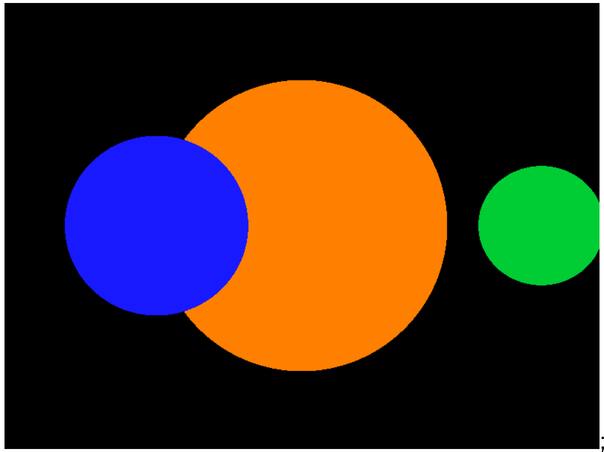
Let's add some colors to the spheres.

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

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


colors

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


snap1.ppm



;

adding lights

We want to add some lights to the world.

Lights have a position and a color

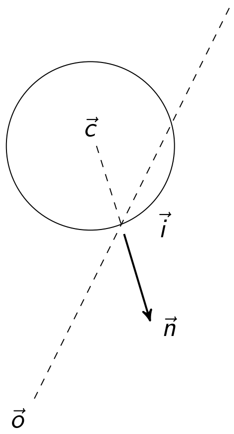
The color of an intersection point is determined by the color of the object combined with the colors from the lights.

Things are getting interesting.

- **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}`

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.

extend Object protocol

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


  defimpl Object do
    def intersect(sphere, ray) do
      Sphere.intersect(sphere, ray)
    end

    def normal(sphere, _, pos) do
      Vector.normalize(Vector.sub(pos, sphere.pos))
    end
  end
```


the world

```
defmodule World do

  @background {0, 0, 0}
  @ambient {0.3, 0.3, 0.3}

  defstruct(objects: [],
            lights: [],
            background: @background,
            ambient: @ambient)
end
```

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

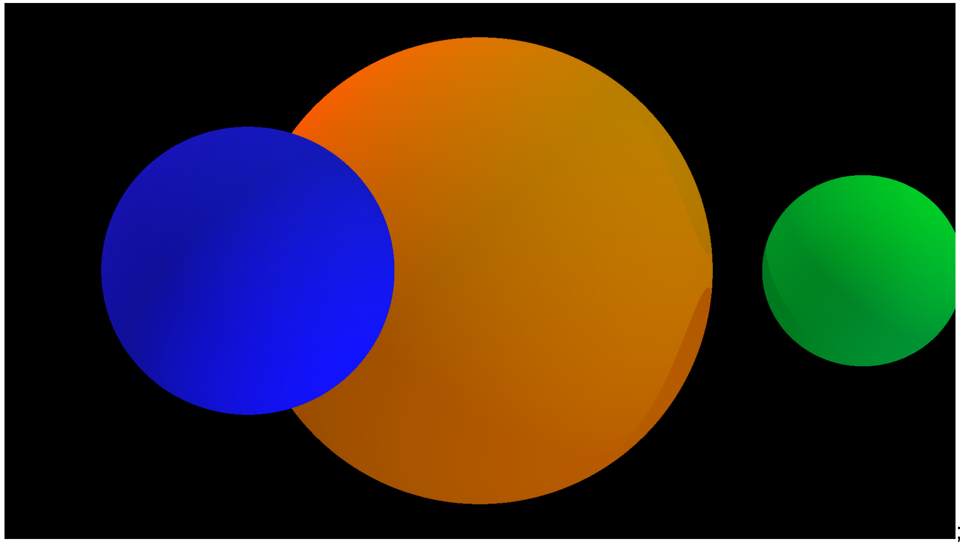
calculating the color

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

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

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

snap2.ppm



;

the fun part

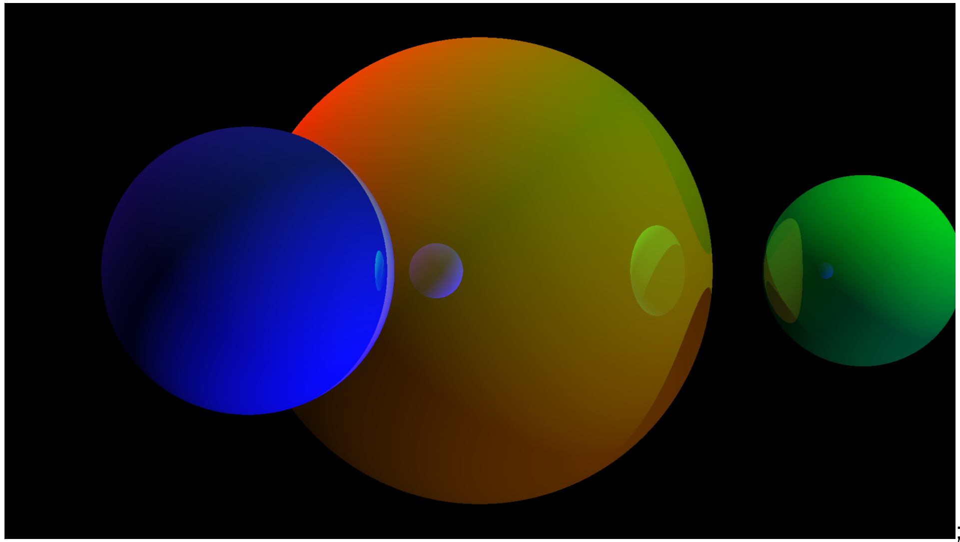
The color of an intersection point depends on:

- color of the object
- combination of light sources
- reflection from other objects

the recursive call

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

defp trace(ray, depth, world) do
  case intersect(world.objects) do
      :
    {d, obj} ->
        :
       reflection = trace(r, depth - 1, world)
       Light.illuminate(obj, reflection, illumination, world)
  end
end
```


snap3.ppm



;

what more

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 (structs are only halfway)
- can we add a new object without rewriting the tracer