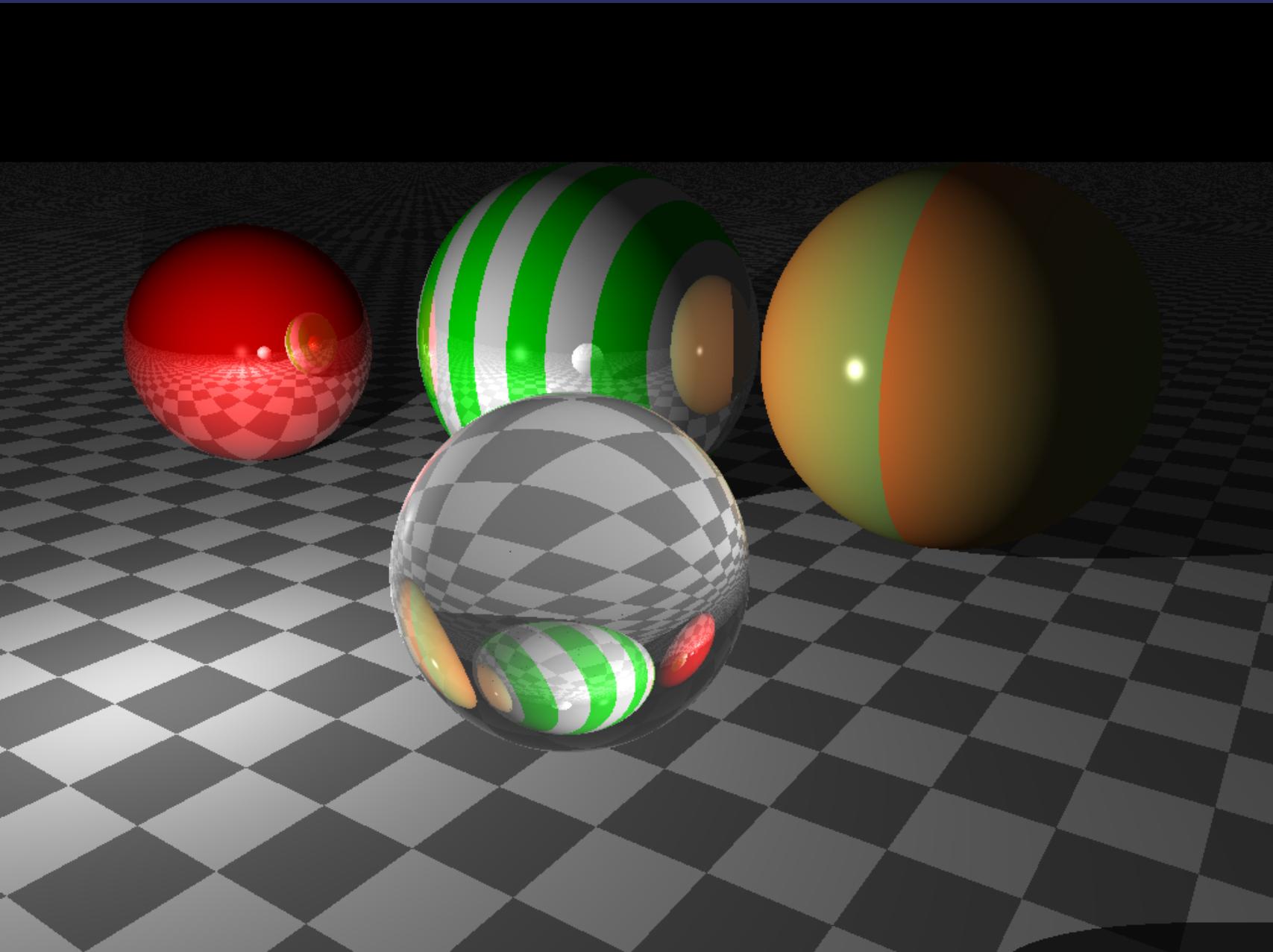


# Environmental effects

## A ray tracing exercise

Functional Scala  
London - 12 Dec 2019



# Agenda

# Agenda

1. Build Ray Tracer components

# Agenda

1. Build Ray Tracer components
2. Test Ray Tracer components

# Agenda

1. Build Ray Tracer components
2. Test Ray Tracer components
3. Wiring things together

# Agenda

1. Build Ray Tracer components
2. Test Ray Tracer components
3. Wiring things together
4. Improving rendering

# ZIO-101

ZIO[-R, +E, +A]

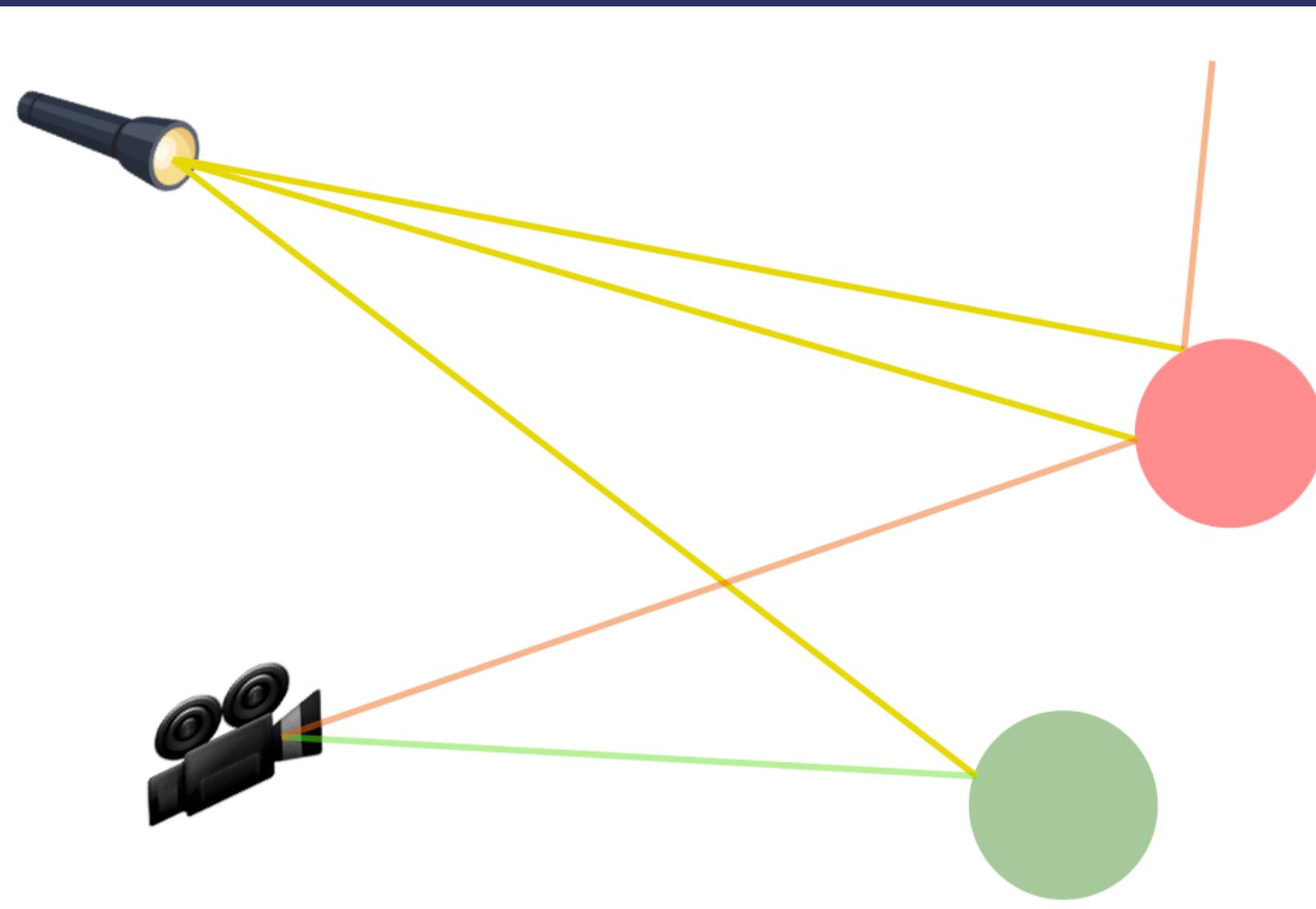


R => IO[Either[E, A]]

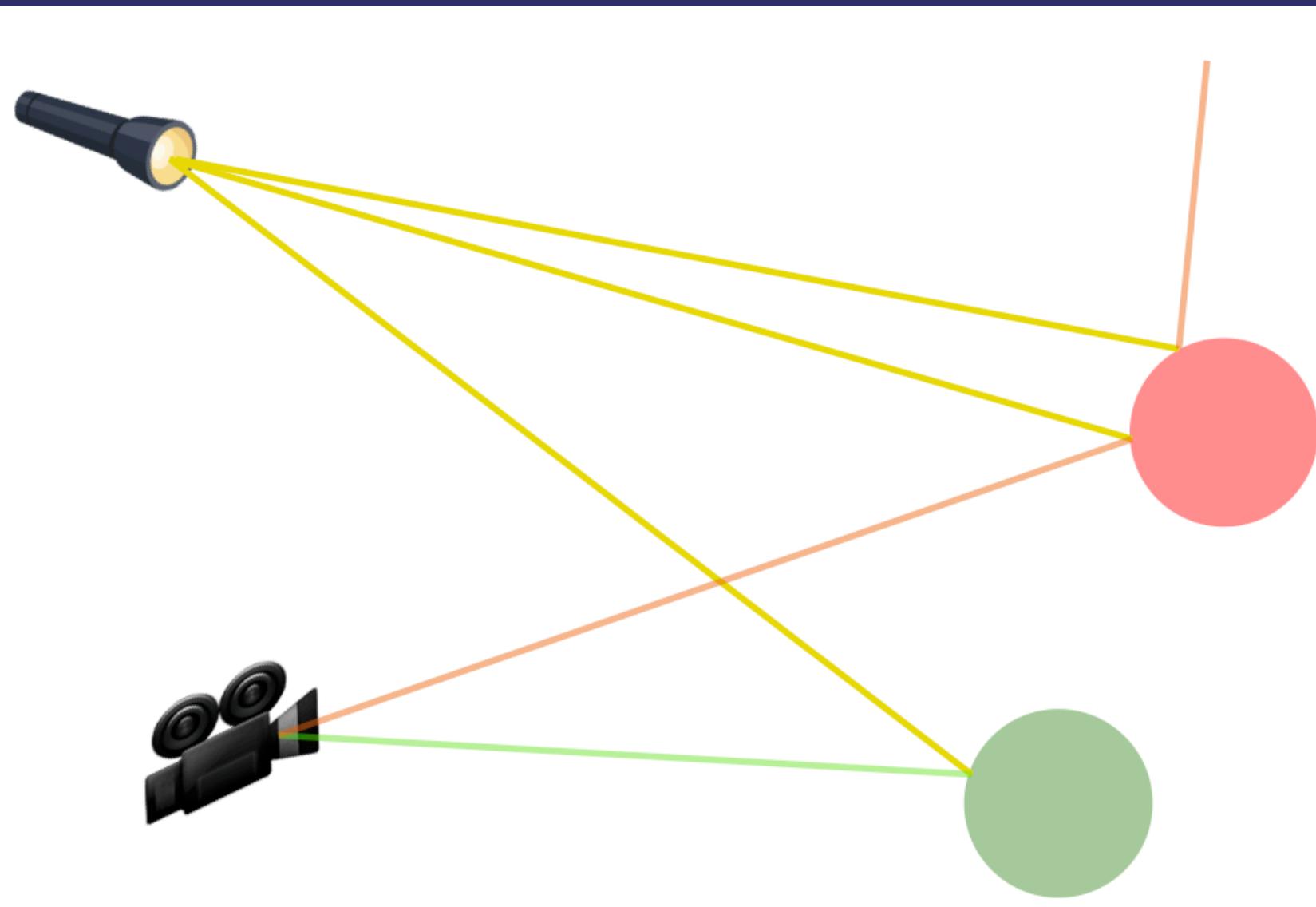


R => Either[E, A]

# Ray tracing

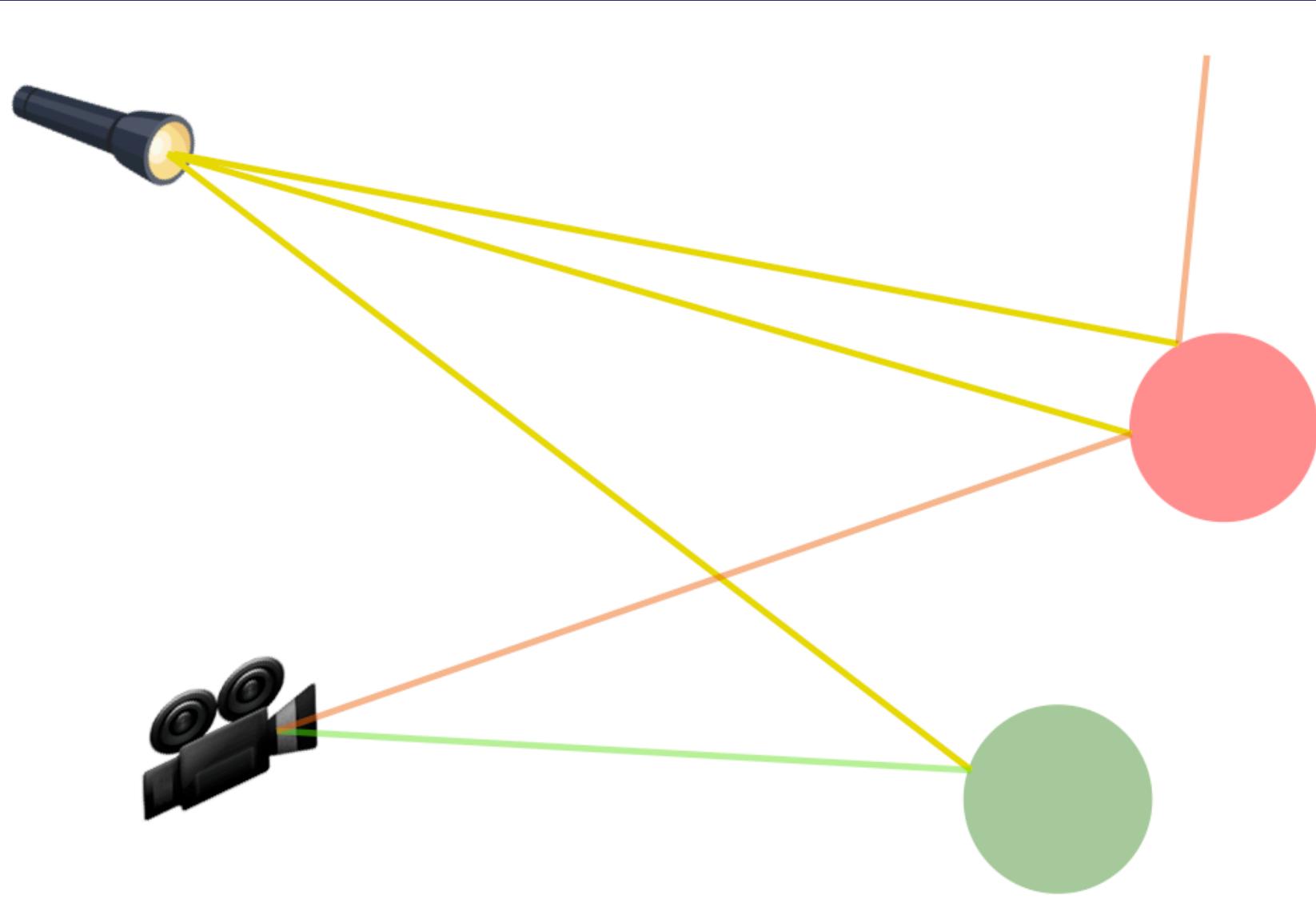


# Ray tracing



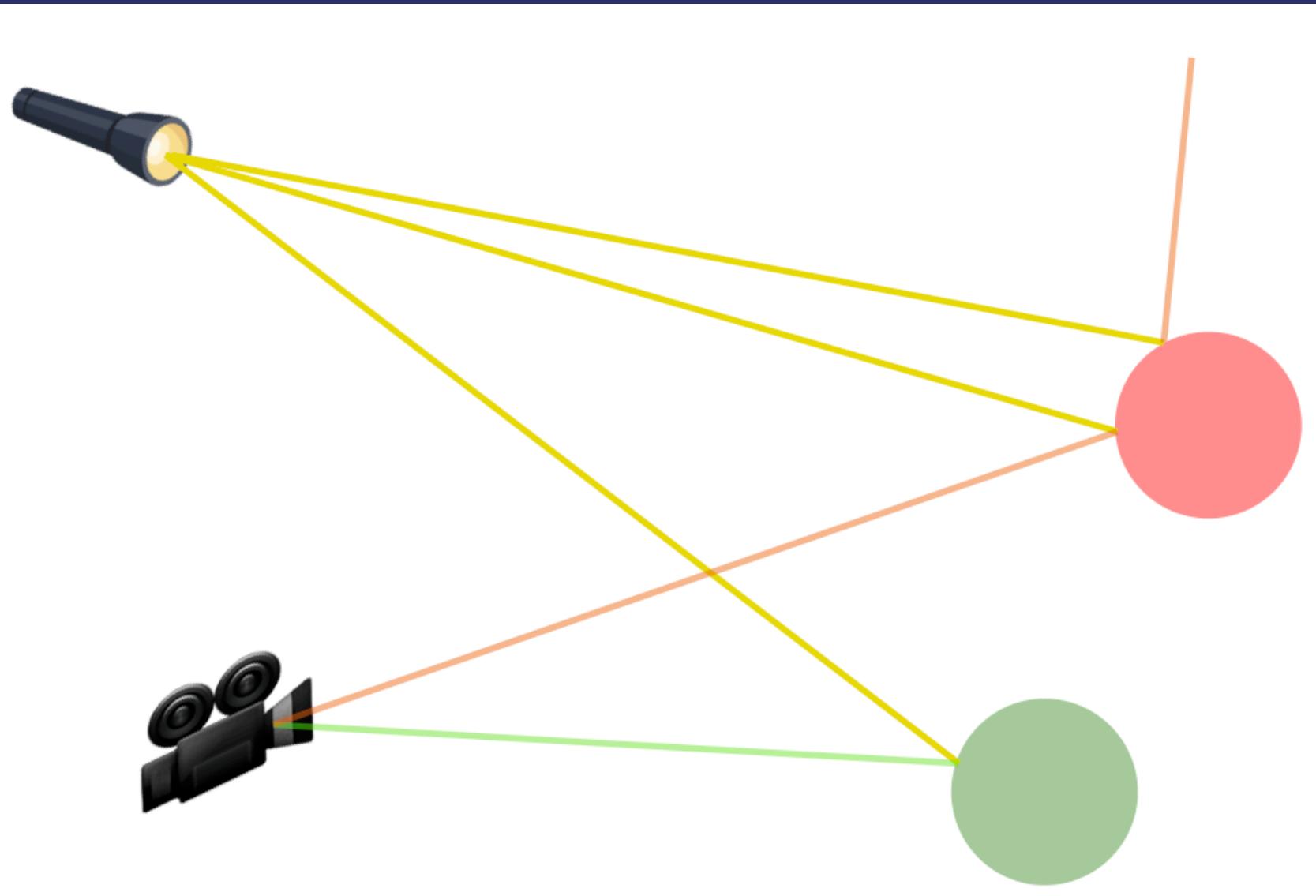
→ World (spheres), light source, camera

# Ray tracing



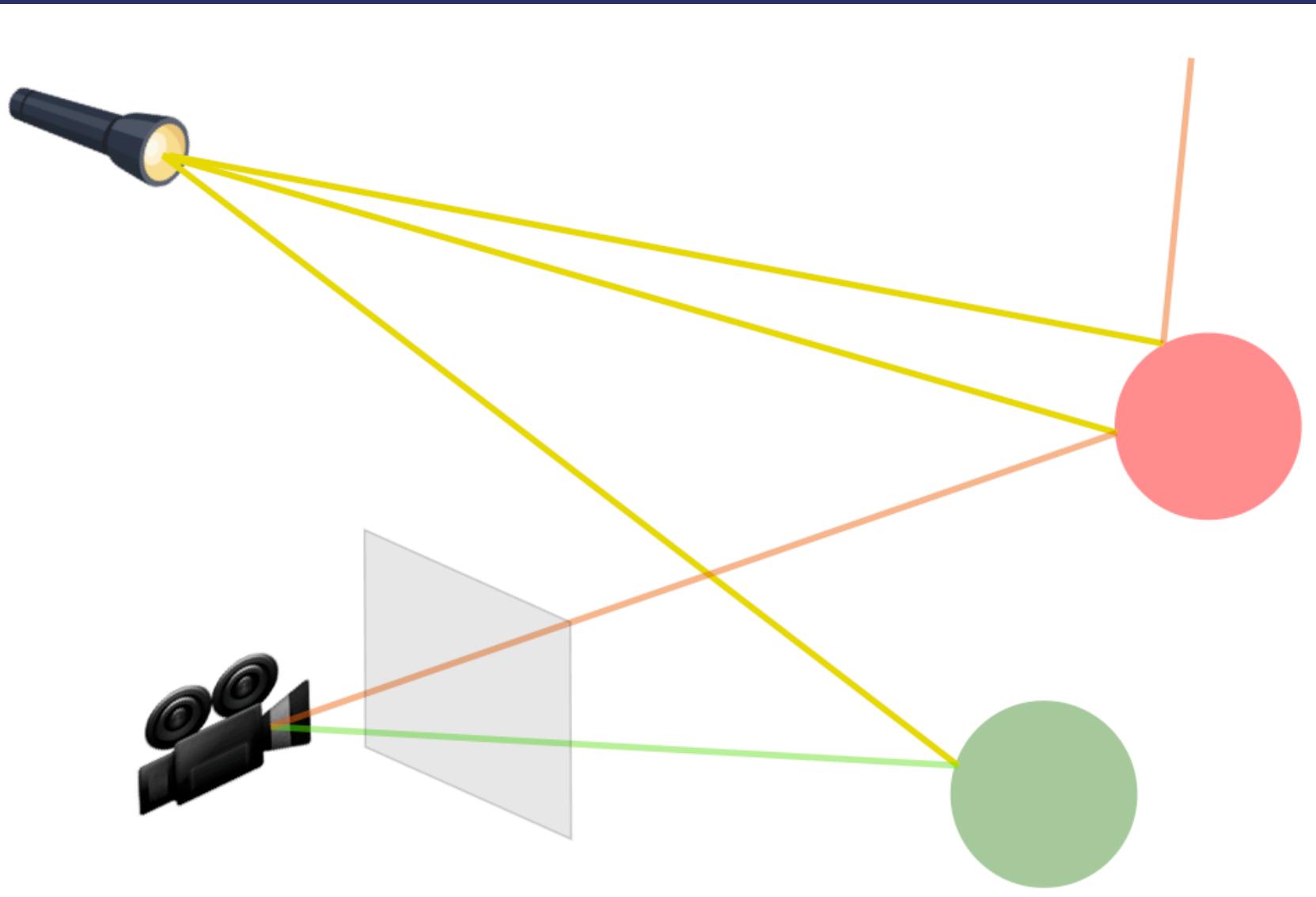
- World (spheres), light source, camera
- Incident rays

# Ray tracing



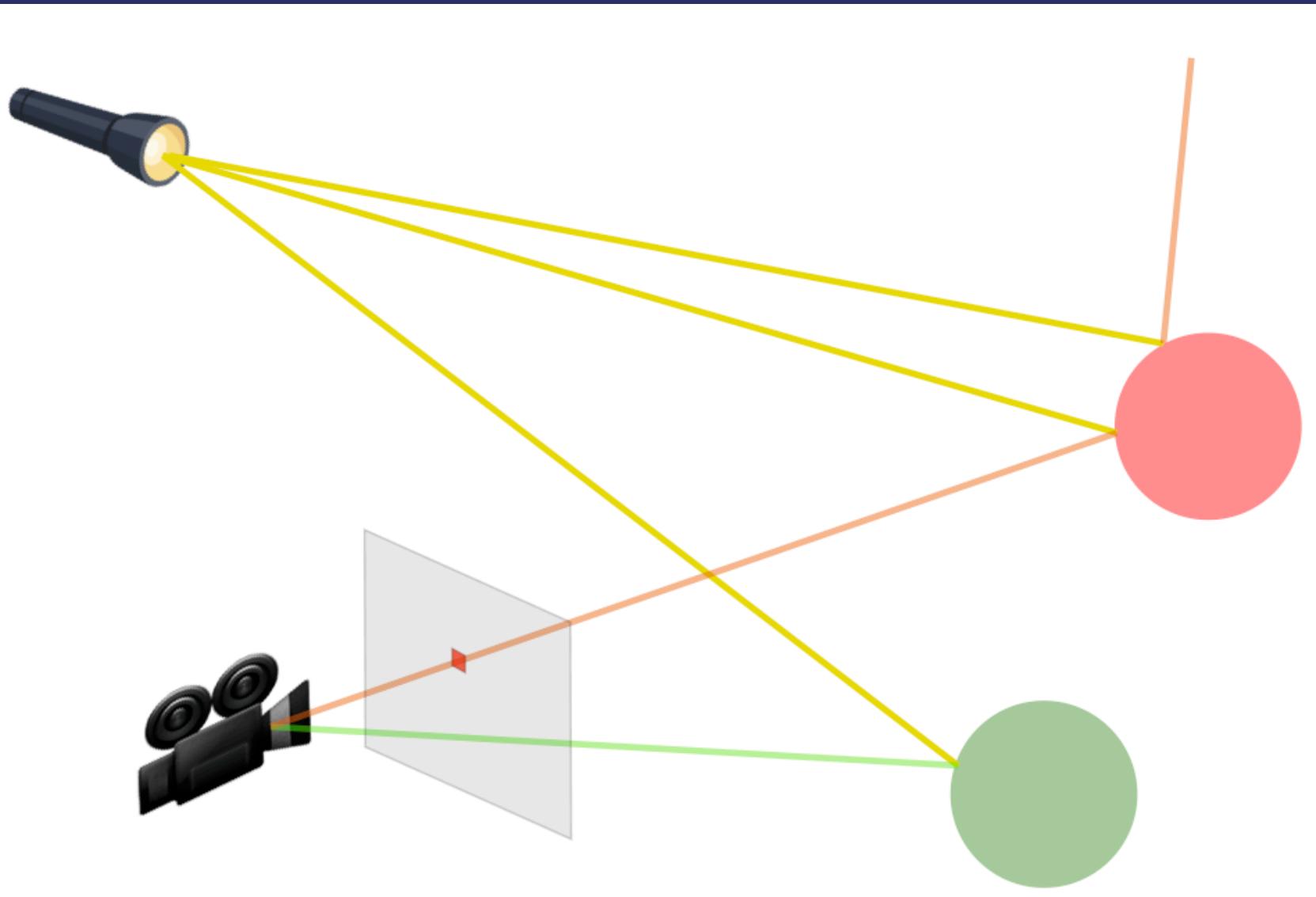
- World (spheres), light source, camera
- Incident rays
- Reflected rays

# Ray tracing



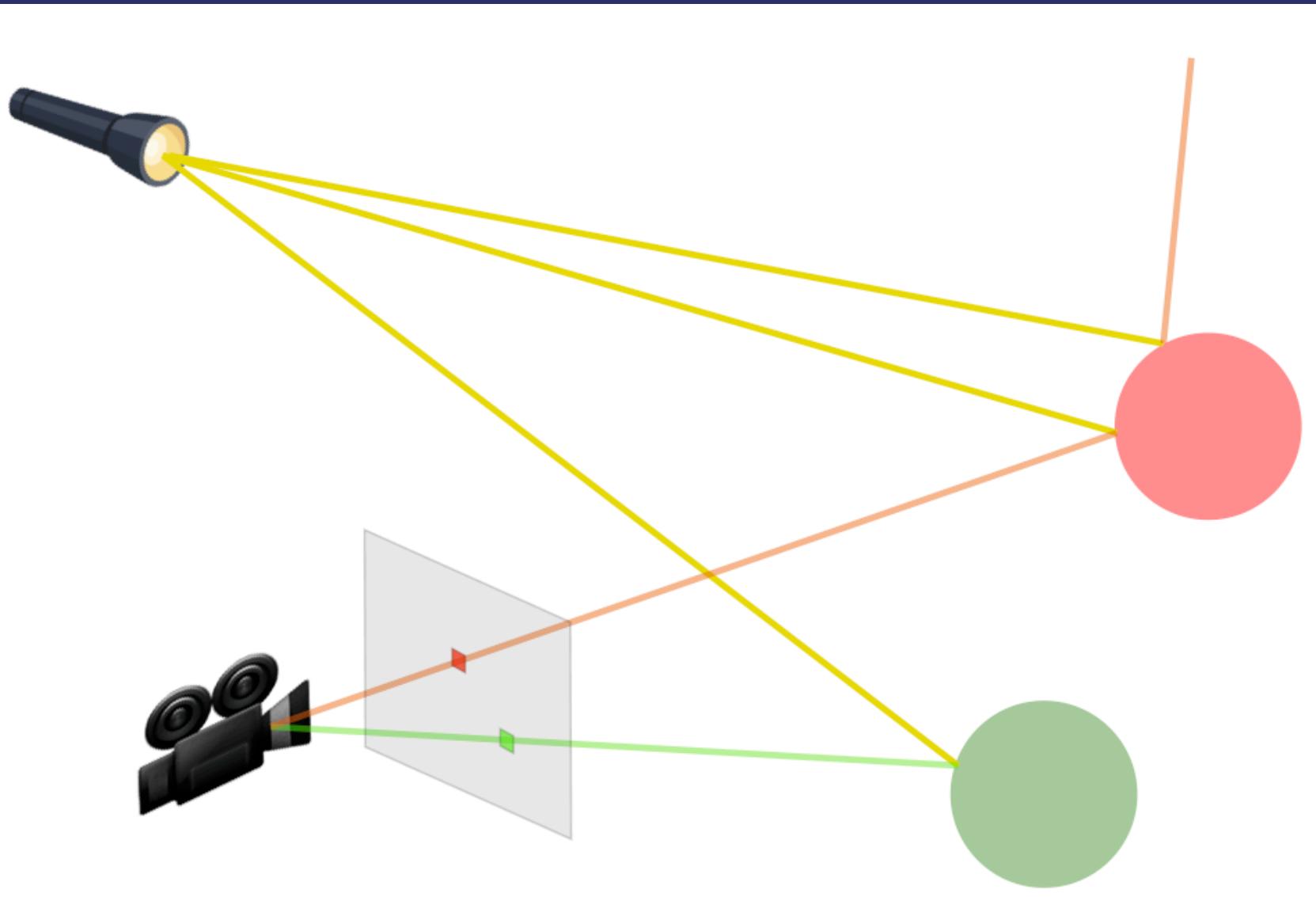
- World (spheres), light source, camera
- Incident rays
- Reflected rays
- Discarded rays
- Canvas

# Ray tracing



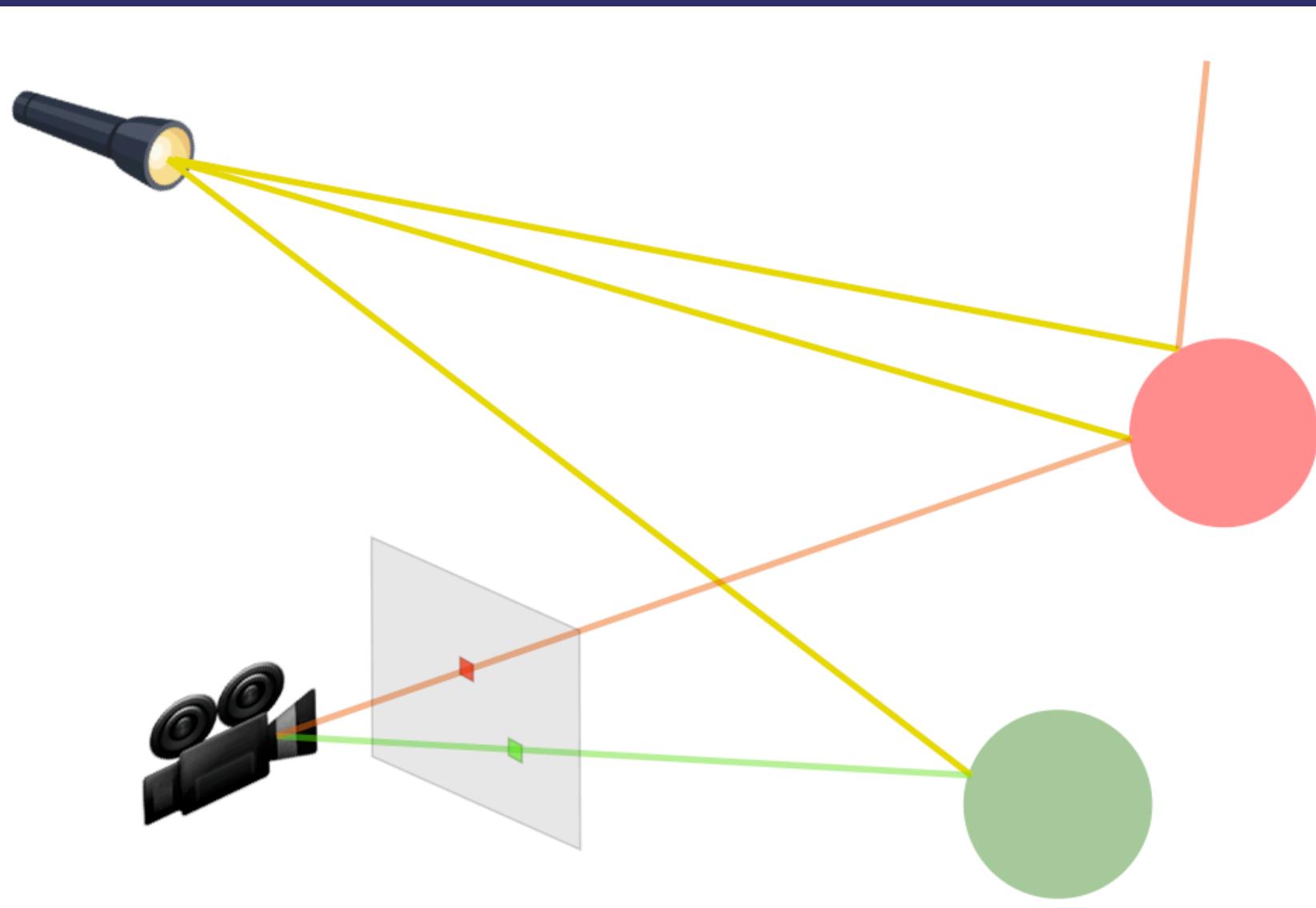
- World (spheres), light source, camera
- Incident rays
- Reflected rays
- Discarded rays
- Canvas
- Colored pixels

# Ray tracing



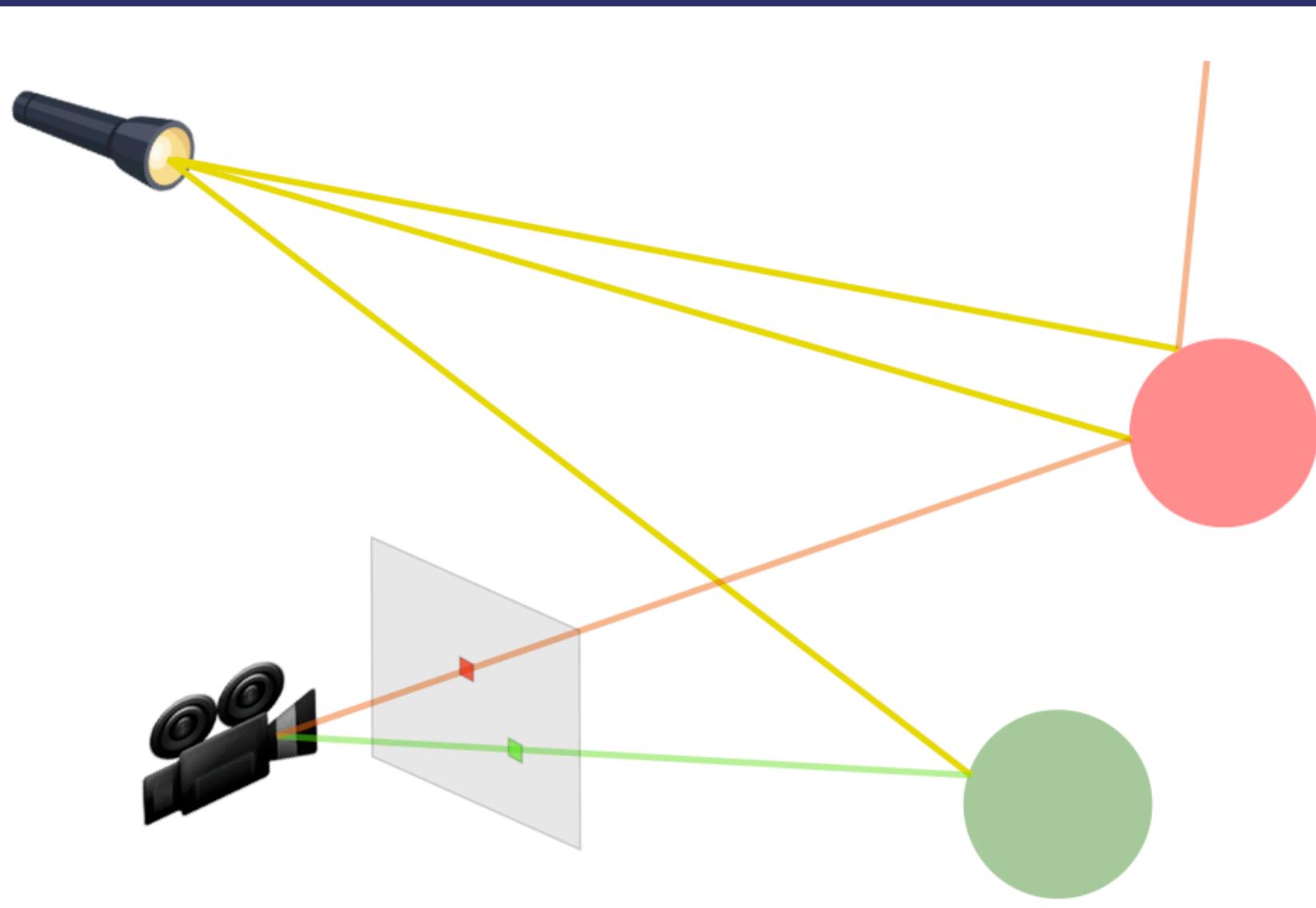
- World (spheres), light source, camera
- Incident rays
- Reflected rays
- Discarded rays
- Canvas
- Colored pixels

# Ray tracing



Options:

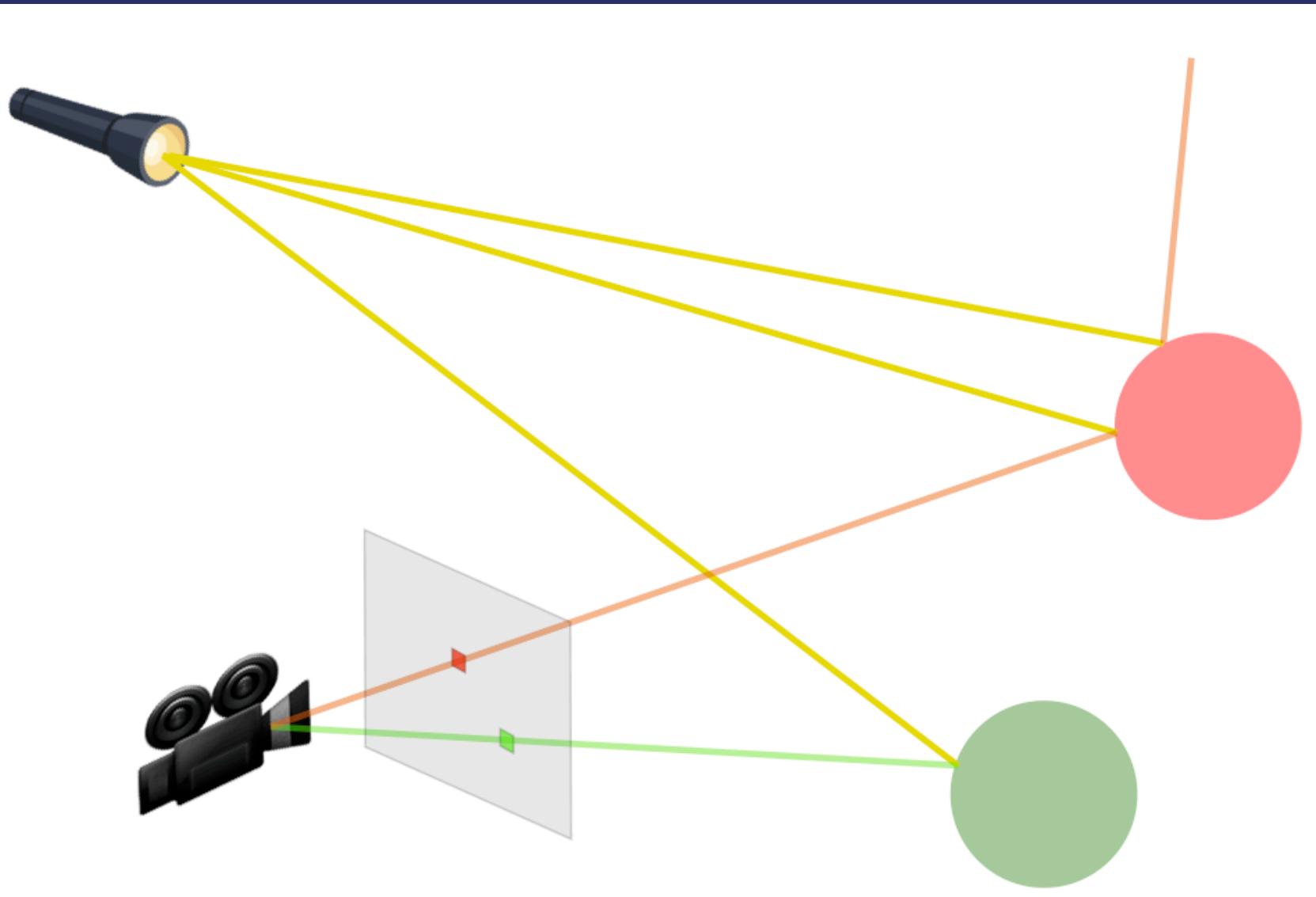
# Ray tracing



Options:

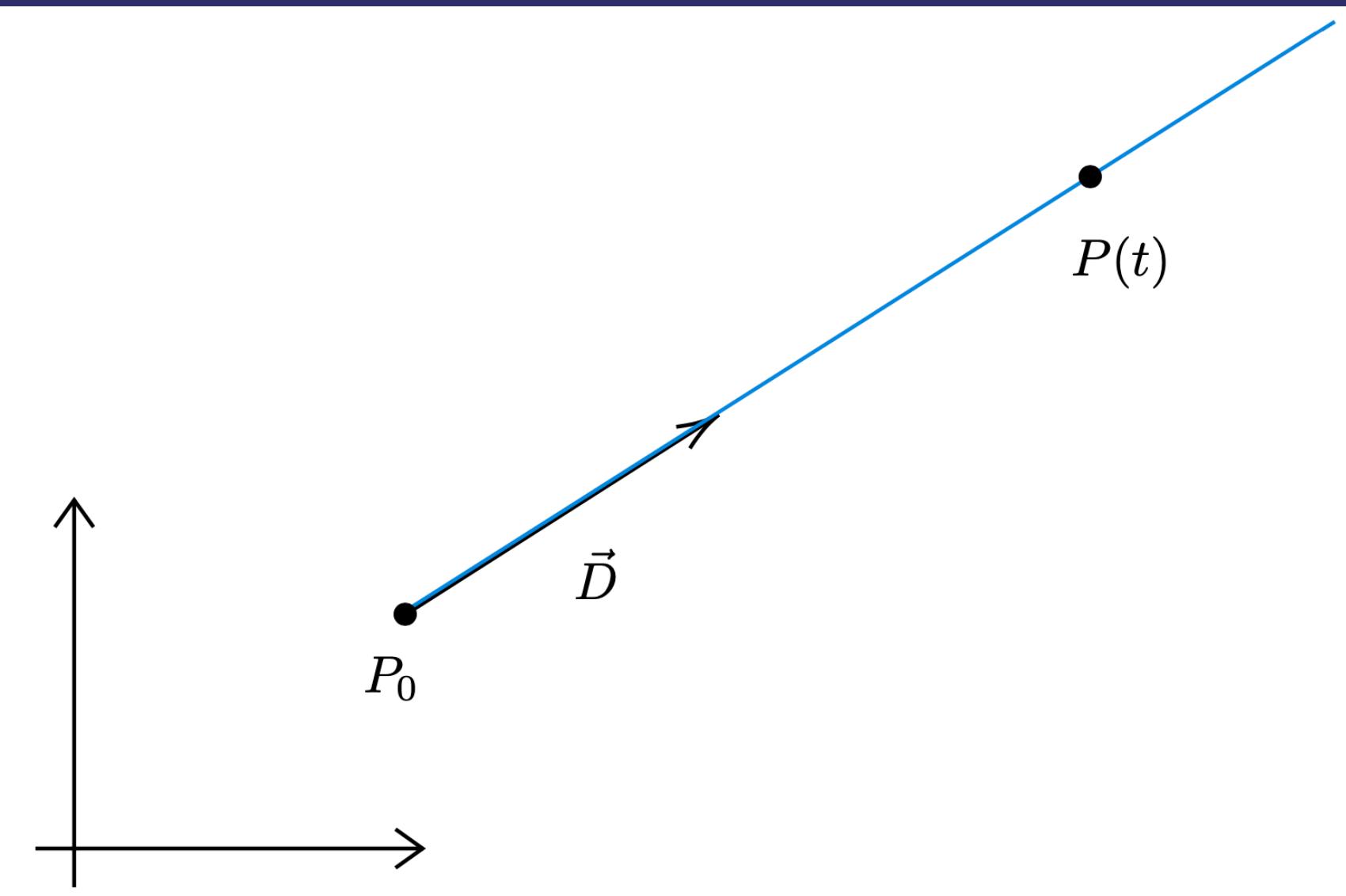
1. Compute all the rays (and discard most of them)

# Ray tracing



Options:

1. Compute all the rays (and discard most of them)
2. Compute only the rays outgoing from the camera through the canvas, and determine how they behave on the surfaces



## Ray

$$P(t) = P_0 + t\vec{D}, t > 0$$

```
case class Ray(origin: Pt, direction: Vec) {  
    def positionAt(t: Double): Pt =  
        origin + (direction * t)  
}
```

# Transformations Module

```
trait AT
/* Module */
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }

  /* Accessor */
  object > extends Service[ATModule] {
    def applyTf(tf: AT, vec: Vec): ZIO[ATModule, ATError, Vec] =
      ZIO.accessM(_.aTModule.applyTf(tf, vec))
    def applyTf(tf: AT, pt: Pt): ZIO[ATModule, ATError, Pt] =
      ZIO.accessM(_.aTModule.applyTf(tf, pt))
    def compose(first: AT, second: AT): ZIO[ATModule, ATError, AT] =
      ZIO.accessM(_.aTModule.compose(first, second))
  }
}
```

# Transformations Module

```
trait AT
/* Module */
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }

  /* Accessor */
  object > extends Service[ATModule] {
    def applyTf(tf: AT, vec: Vec): ZIO[ATModule, ATError, Vec] =
      ZIO.accessM(_.aTModule.applyTf(tf, vec))
    def applyTf(tf: AT, pt: Pt): ZIO[ATModule, ATError, Pt] =
      ZIO.accessM(_.aTModule.applyTf(tf, pt))
    def compose(first: AT, second: AT): ZIO[ATModule, ATError, AT] =
      ZIO.accessM(_.aTModule.compose(first, second))
  }
}
```

# Transformations Module

```
trait AT
/* Module */
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }

  /* Accessor */
  object > extends Service[ATModule] {
    def applyTf(tf: AT, vec: Vec): ZIO[ATModule, ATError, Vec] =
      ZIO.accessM(_.aTModule.applyTf(tf, vec))
    def applyTf(tf: AT, pt: Pt): ZIO[ATModule, ATError, Pt] =
      ZIO.accessM(_.aTModule.applyTf(tf, pt))
    def compose(first: AT, second: AT): ZIO[ATModule, ATError, AT] =
      ZIO.accessM(_.aTModule.compose(first, second))
  }
}
```

# Transformations Module

```
trait AT
/* Module */
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }

  /* Accessor */
  object > extends Service[ATModule] {
    def applyTf(tf: AT, vec: Vec): ZIO[ATModule, ATError, Vec] =
      ZIO.accessM(_.aTModule.applyTf(tf, vec))
    def applyTf(tf: AT, pt: Pt): ZIO[ATModule, ATError, Pt] =
      ZIO.accessM(_.aTModule.applyTf(tf, pt))
    def compose(first: AT, second: AT): ZIO[ATModule, ATError, AT] =
      ZIO.accessM(_.aTModule.compose(first, second))
  }
}
```

# Transformations Module

```
import zio.macros.annotation.accessible

trait AT
/* Module */
@accessible(">")
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }

  /* Accessor is generated
  object > extends Service[ATModule] {
    def applyTf(tf: AT, vec: Vec): ZIO[ATModule, ATError, Vec] =
      ZIO.accessM(_.aTModule.applyTf(tf, vec))
    def applyTf(tf: AT, pt: Pt): ZIO[ATModule, ATError, Pt] =
      ZIO.accessM(_.aTModule.applyTf(tf, pt))
    def compose(first: AT, second: AT): ZIO[ATModule, ATError, AT] =
      ZIO.accessM(_.aTModule.compose(first, second))
  }
  */
}
```

# Transformations Module

```
trait AT
/* Module */
@accessible(">")
trait ATModule {
  val aTModule: ATModule.Service[Any]
}

object ATModule {
  /* Service */
  trait Service[R] {
    def applyTf(tf: AT, vec: Vec): ZIO[R, ATError, Vec]
    def applyTf(tf: AT, pt: Pt): ZIO[R, ATError, Pt]
    def compose(first: AT, second: AT): ZIO[R, ATError, AT]
  }
}
```

# Transformations Module

```
val rotatedPt =  
  for {  
    rotateX <- ATModule.>.rotateX(math.Pi / 2)  
    _       <- Log.>.info("rotated of π/2")  
    res     <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
  } yield res
```

# Transformations Module

```
val rotatedPt: ZIO[ATModule with Log, ATError, Pt] =  
for {  
    rotateX <- ATModule.>.rotateX(math.Pi / 2)  
    _           <- Log.>.info("rotated of π/2")  
    res        <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
} yield res
```

# Transformations Module - Live

```
val rotated: ZIO[ATModule, ATError, Vec] =  
  for {  
    rotateX <- ATModule.>.rotateX(math.Pi/2)  
    res      <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
  } yield res
```

# Transformations Module - Live

```
val rotated: ZIO[ATModule, ATError, Vec] =  
  for {  
    rotateX <- ATModule.>.rotateX(math.Pi/2)  
    res      <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
  } yield res
```

$$\rightarrow \text{Vec}(x, y, z) \Rightarrow [x, y, z, 0]^T$$

# Transformations Module - Live

```
val rotated: ZIO[ATModule, ATError, Vec] =  
  for {  
    rotateX <- ATModule.>.rotateX(math.Pi/2)  
    res      <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
  } yield res
```

$\rightarrow \text{Vec}(x, y, z) \Rightarrow [x, y, z, 0]^T$   
 $\rightarrow \text{Pt}(x, y, z) \Rightarrow [x, y, z, 1]^T$

# Transformations Module - Live

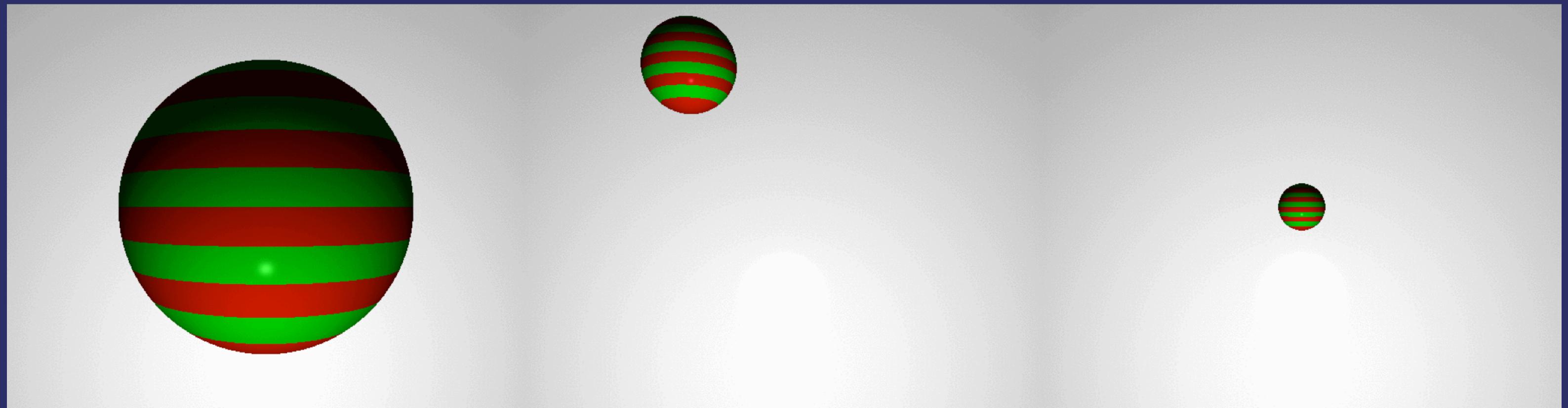
```
val rotated: ZIO[ATModule, ATError, Vec] =  
  for {  
    rotateX <- ATModule.>.rotateX(math.Pi/2)  
    res      <- ATModule.>.applyTf(rotateX, Pt(1, 1, 1))  
  } yield res
```

$$\rightarrow \text{Vec}(x, y, z) \Rightarrow [x, y, z, 0]^T$$

$$\rightarrow \text{Pt}(x, y, z) \Rightarrow [x, y, z, 1]^T$$

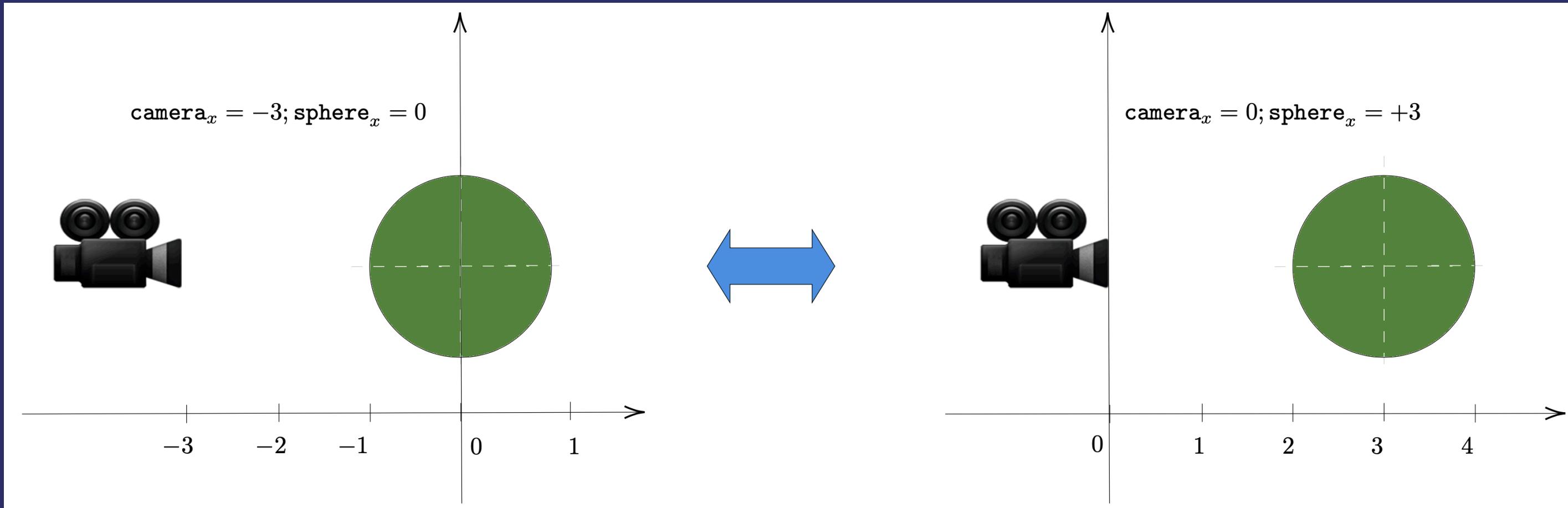
$$\rightarrow \text{rotated} = \begin{pmatrix} \cos \pi/2 & -\sin \pi/2 & 0 & 0 \\ \sin \pi/2 & \cos \pi/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

# Layer 1: Transformations



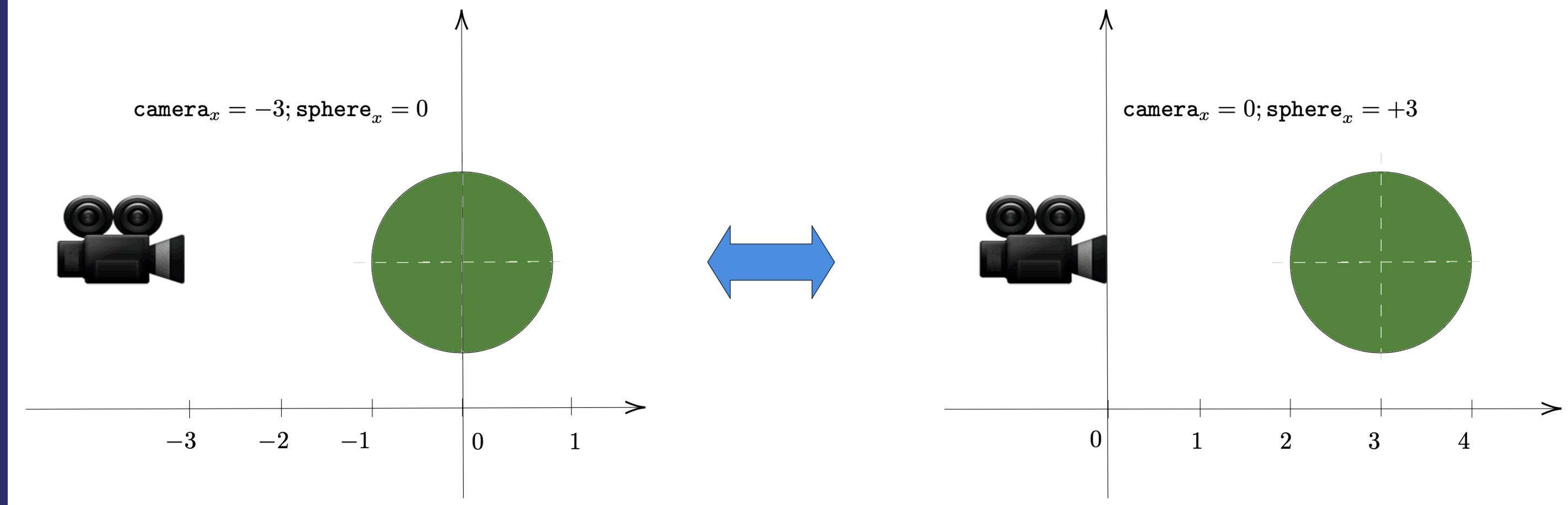
# Camera

**Everything is relative!**



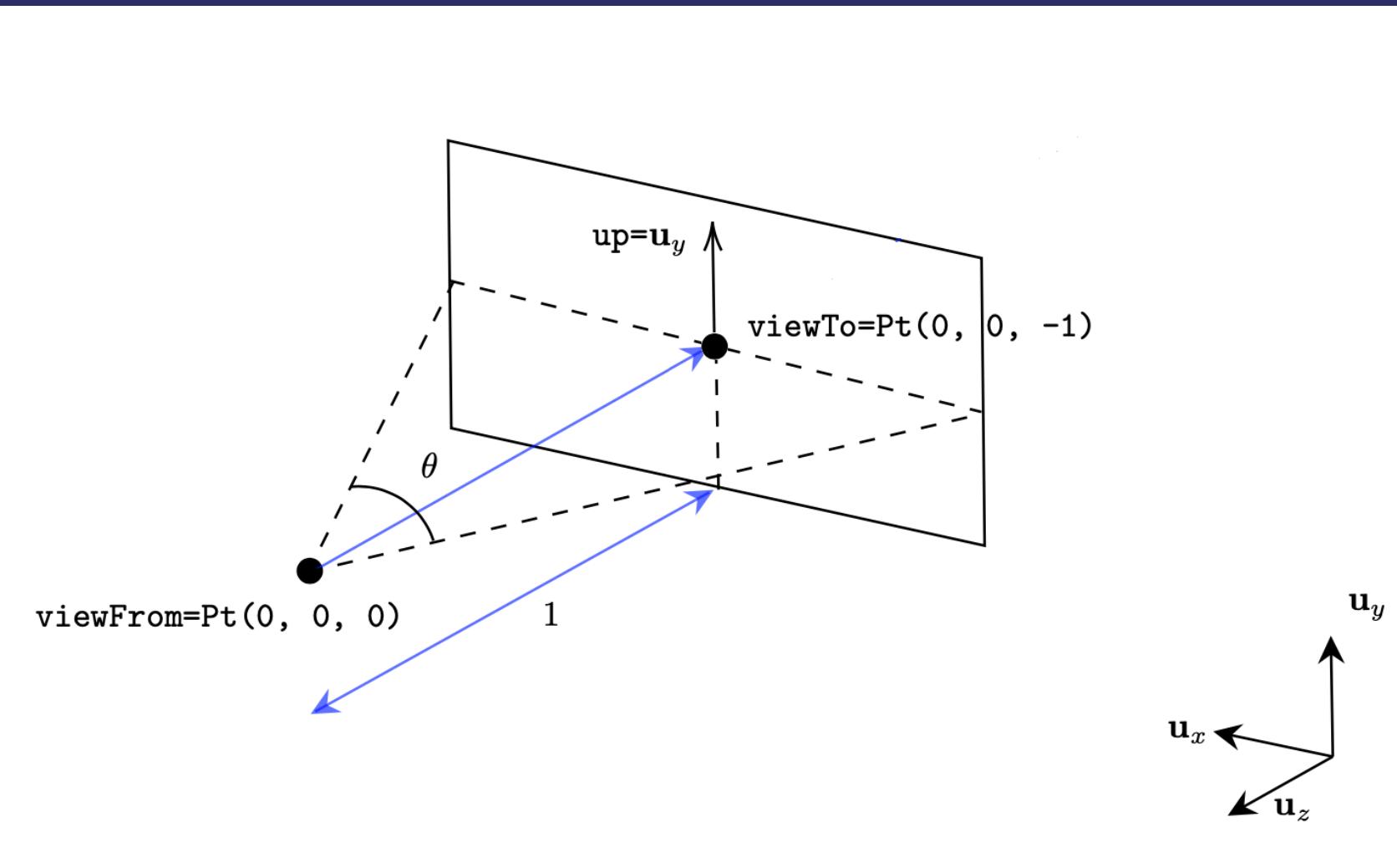
# Camera

**Everything is relative!**



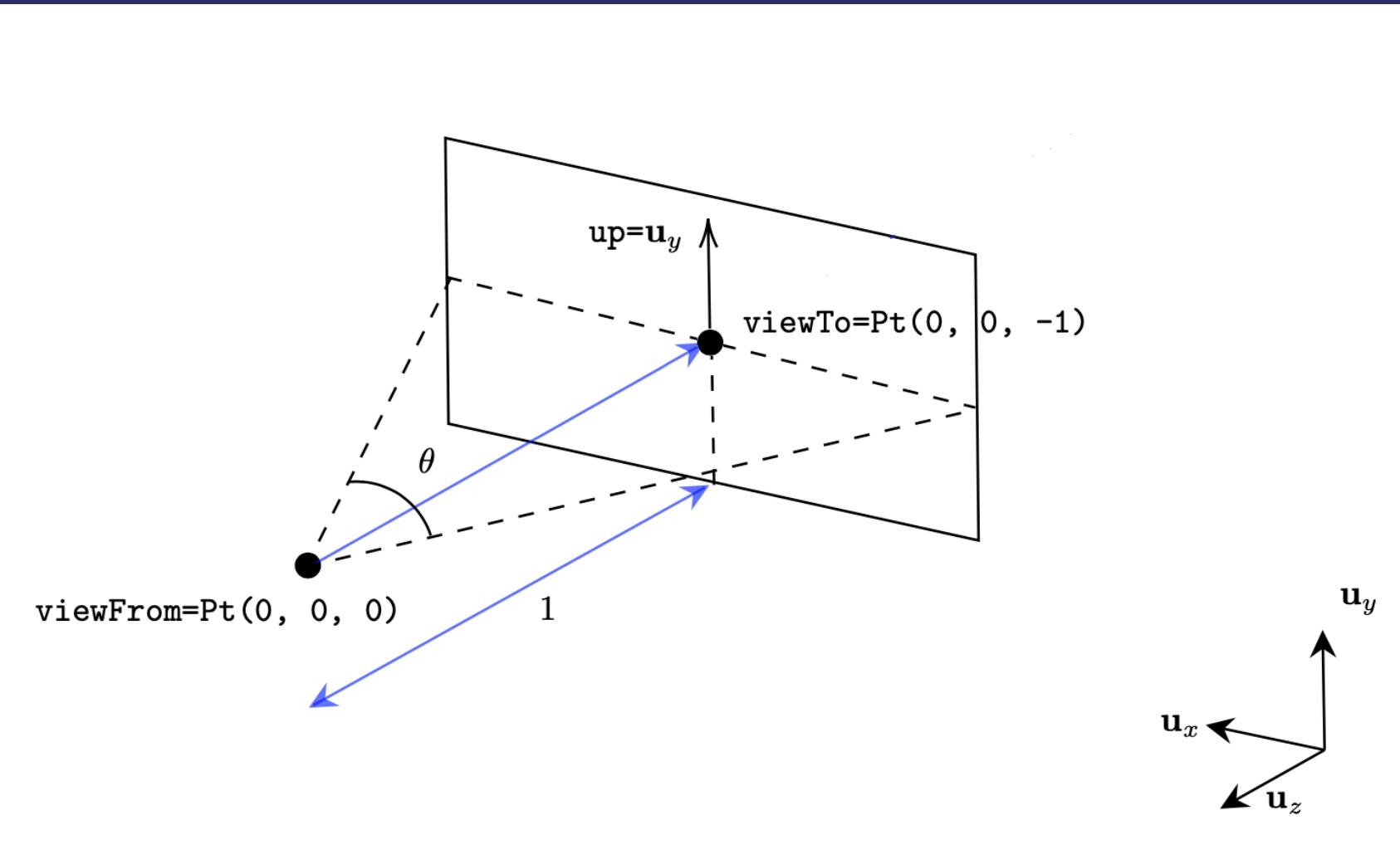
→ Canonical camera: observe always from  $x = 0$  and translate the world by +3

# Camera - canonical



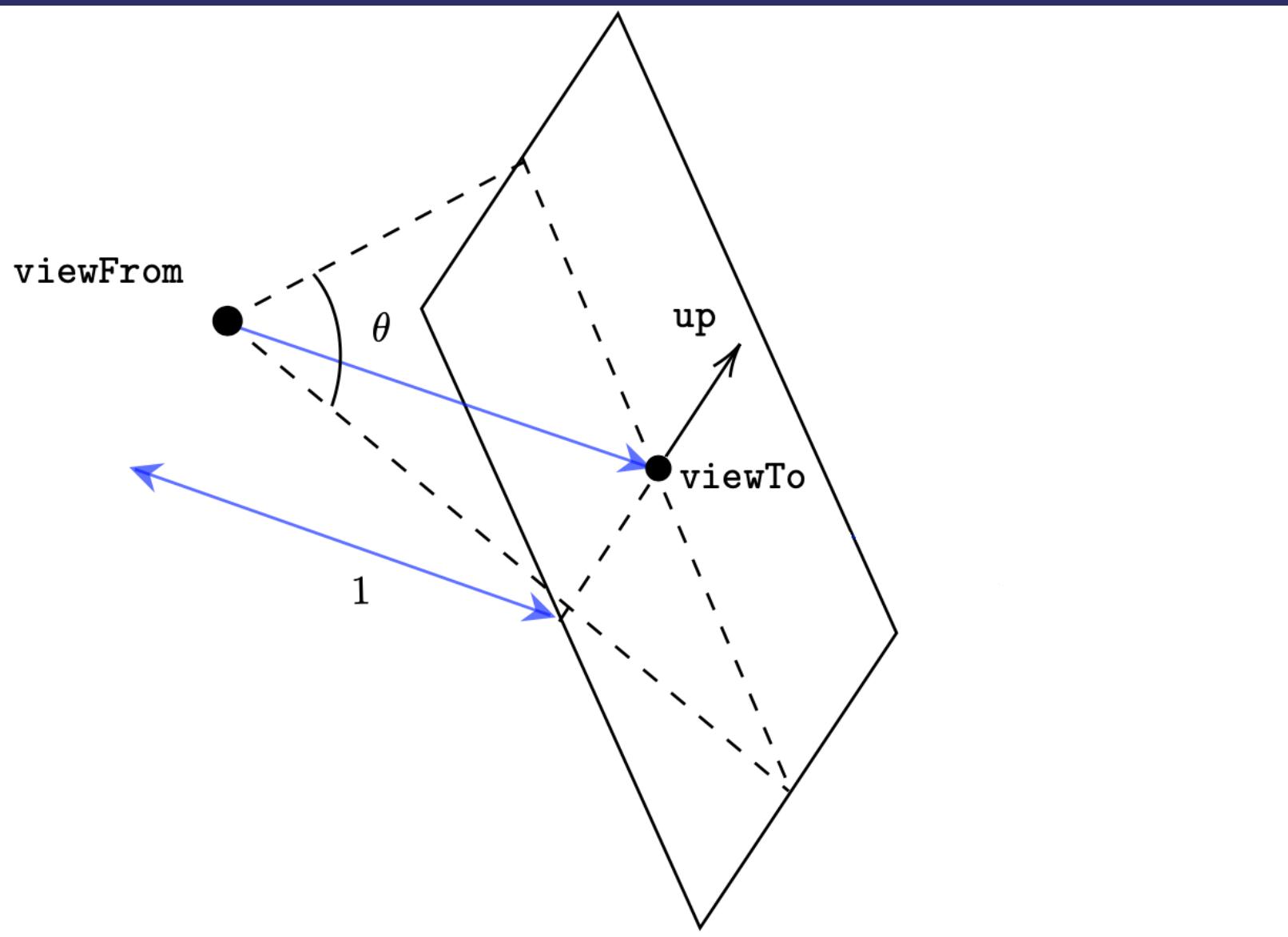
```
case class Camera (  
    hRes: Int,  
    vRes: Int,  
    fieldOfViewRad: Double,  
    tf: AT // world transformation  
)
```

# Camera - canonical



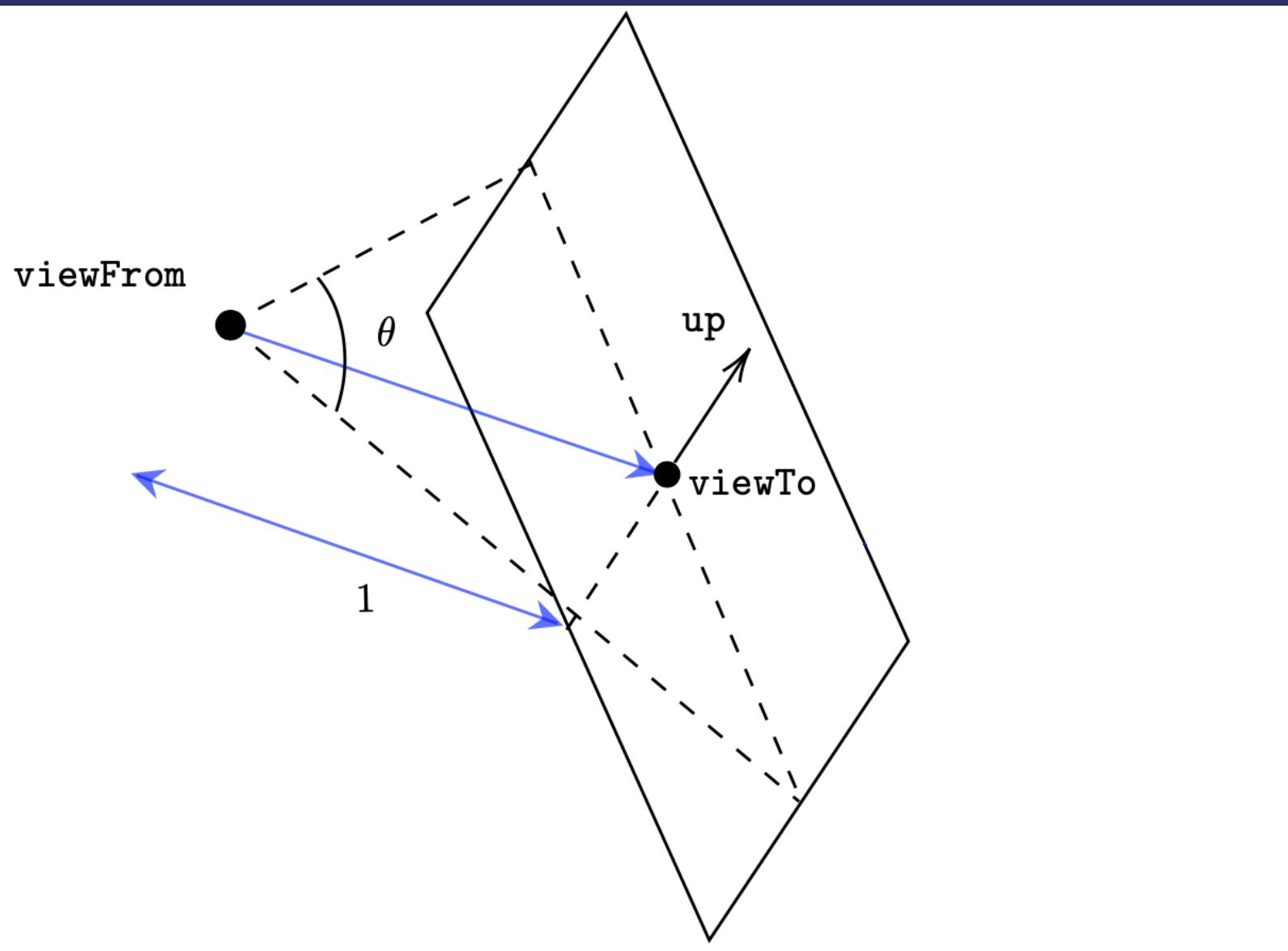
```
case class Camera (  
    hRes: Int,  
    vRes: Int,  
    fieldOfViewRad: Double,  
    tf: AT // world transformation  
)
```

# Camera - generic



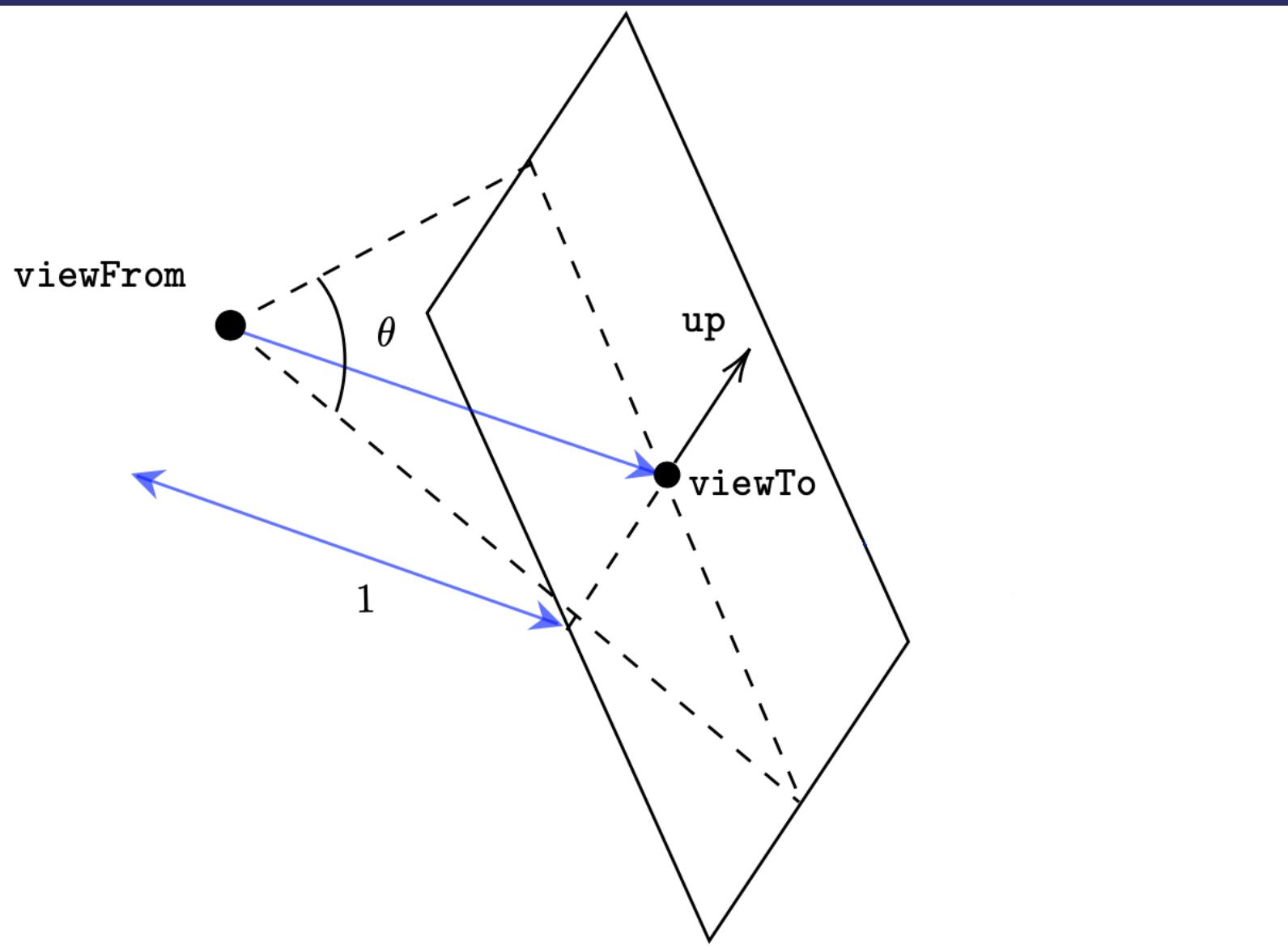
```
object Camera {  
  def make(  
    viewFrom: Pt,  
    viewTo: Pt,  
    upDirection: Vec,  
    visualAngleRad: Double,  
    hRes: Int,  
    vRes: Int):  
    ZIO[ATModule, AlgebraicError, Camera] =  
    worldTransformation(viewFrom, viewTo, upDirection).map {  
      worldTf => Camera(hRes, vRes, visualAngleRad, worldTf)  
    }  
}
```

# Camera - generic



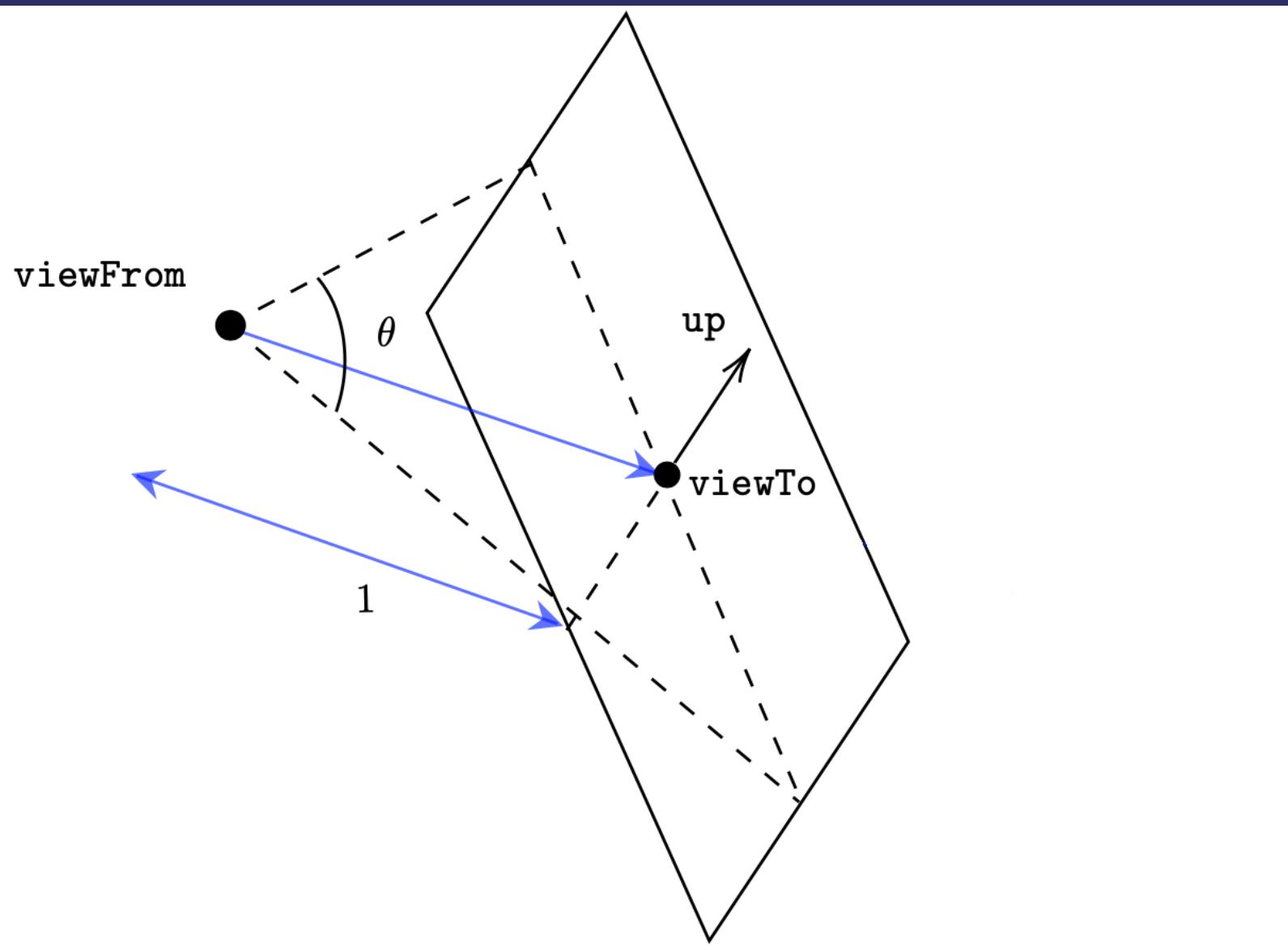
```
object Camera {  
  def make(  
    viewFrom: Pt,  
    viewTo: Pt,  
    upDirection: Vec,  
    visualAngleRad: Double,  
    hRes: Int,  
    vRes: Int):  
    ZIO[ATModule, AlgebraicError, Camera] =  
    worldTransformation(viewFrom, viewTo, upDirection).map {  
      worldTf => Camera(hRes, vRes, visualAngleRad, worldTf)  
    }  
}
```

# Camera - generic



```
object Camera {  
  def make(  
    viewFrom: Pt,  
    viewTo: Pt,  
    upDirection: Vec,  
    visualAngleRad: Double,  
    hRes: Int,  
    vRes: Int):  
    ZIO[ATModule, AlgebraicError, Camera] =  
    worldTransformation(viewFrom, viewTo, upDirection).map {  
      worldTf => Camera(hRes, vRes, visualAngleRad, worldTf)  
    }  
}
```

# Camera - generic



```
object Camera {  
  def make(  
    viewFrom: Pt,  
    viewTo: Pt,  
    upDirection: Vec,  
    visualAngleRad: Double,  
    hRes: Int,  
    vRes: Int):  
    ZIO[ATModule, AlgebraicError, Camera] =  
    worldTransformation(viewFrom, viewTo, upDirection).map {  
      worldTf => Camera(hRes, vRes, visualAngleRad, worldTf)  
    }  
}
```

# World

```
sealed trait Shape {  
    def transformation: AT  
    def material: Material  
}  
  
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

→ Sphere.canonical  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

```
→ Sphere.canonical {(x, y, z) : x2 + y2 + z2 = 1}  
→ Plane.canonical {(x, y, z) : y = 0}
```

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

```
→ Sphere.canonical {(x, y, z) : x2 + y2 + z2 = 1}  
→ Plane.canonical {(x, y, z) : y = 0}
```

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

```
→ Sphere.canonical {(x, y, z) : x2 + y2 + z2 = 1}  
→ Plane.canonical {(x, y, z) : y = 0}
```

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

```
→ Sphere.canonical {(x, y, z) : x2 + y2 + z2 = 1}  
→ Plane.canonical {(x, y, z) : y = 0}
```

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

```
→ Sphere.canonical {(x, y, z) : x2 + y2 + z2 = 1}  
→ Plane.canonical {(x, y, z) : y = 0}
```

```
sealed trait Shape {  
  def transformation: AT  
  def material: Material  
}
```

```
case class Sphere(transformation: AT, material: Material) extends Shape  
case class Plane(transformation: AT, material: Material) extends Shape
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale     <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed  <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale      <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed   <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale      <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed   <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale      <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed   <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale      <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed   <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

# World

## Make a world

```
object Sphere {  
    def make(center: Pt, radius: Double, mat: Material): ZIO[ATModule, ATError, Sphere] = for {  
        scale      <- ATModule.>.scale(radius, radius, radius)  
        translate <- ATModule.>.translate(center.x, center.y, center.z)  
        composed   <- ATModule.>.compose(scale, translate)  
    } yield Sphere(composed, mat)  
}  
  
object Plane {  
    def make(...): ZIO[ATModule, ATError, Plane] = ???  
}  
  
case class World(pointLight: PointLight, objects: List[Shape])
```

→ Everything requires ATModule

# World Rendering - Top Down

**Rastering - Generate a stream of colored pixels**

```
@accessible(">")
trait RasteringModule {
  val rasteringModule: RasteringModule.Service[Any]
}

object RasteringModule {
  trait Service[R] {
    def raster(world: World, camera: Camera):
      ZStream[R, RayTracerError, ColoredPixel]
  }
}
```

# World Rendering - Top Down

**Rastering - Generate a stream of colored pixels**

```
@accessible(">")
trait RasteringModule {
  val rasteringModule: RasteringModule.Service[Any]
}

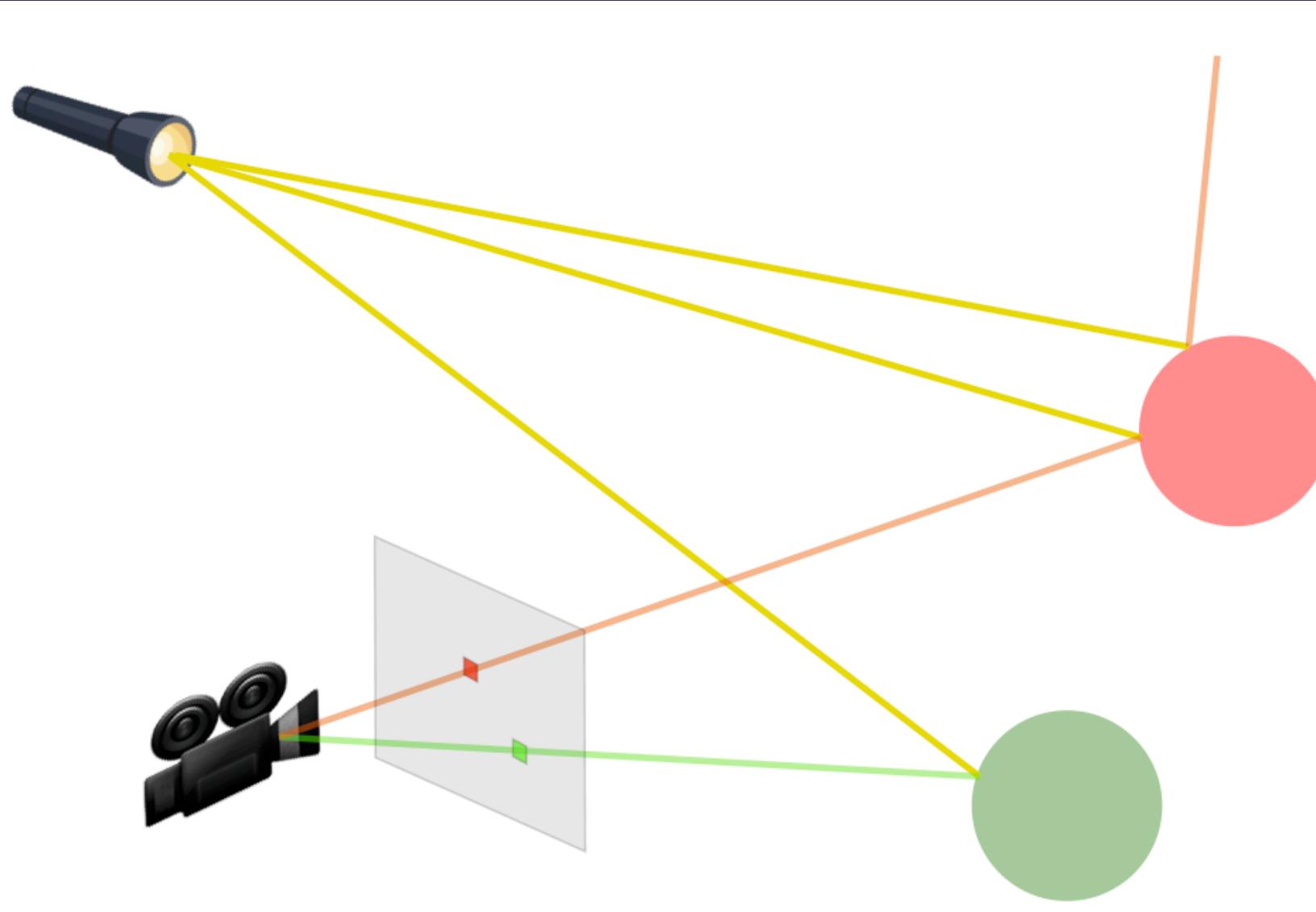
object RasteringModule {
  trait Service[R] {
    def raster(world: World, camera: Camera):
      ZStream[R, RayTracerError, ColoredPixel]
  }
}
```

# World Rendering - Top Down

**Rastering - Generate a stream of colored pixels**

```
@accessible(">")  
trait RasteringModule {  
    val rasteringModule: RasteringModule.Service[Any]  
}  
object RasteringModule {  
    trait Service[R] {  
        def raster(world: World, camera: Camera):  
            ZStream[R, RayTracerError, ColoredPixel]  
    }  
}
```

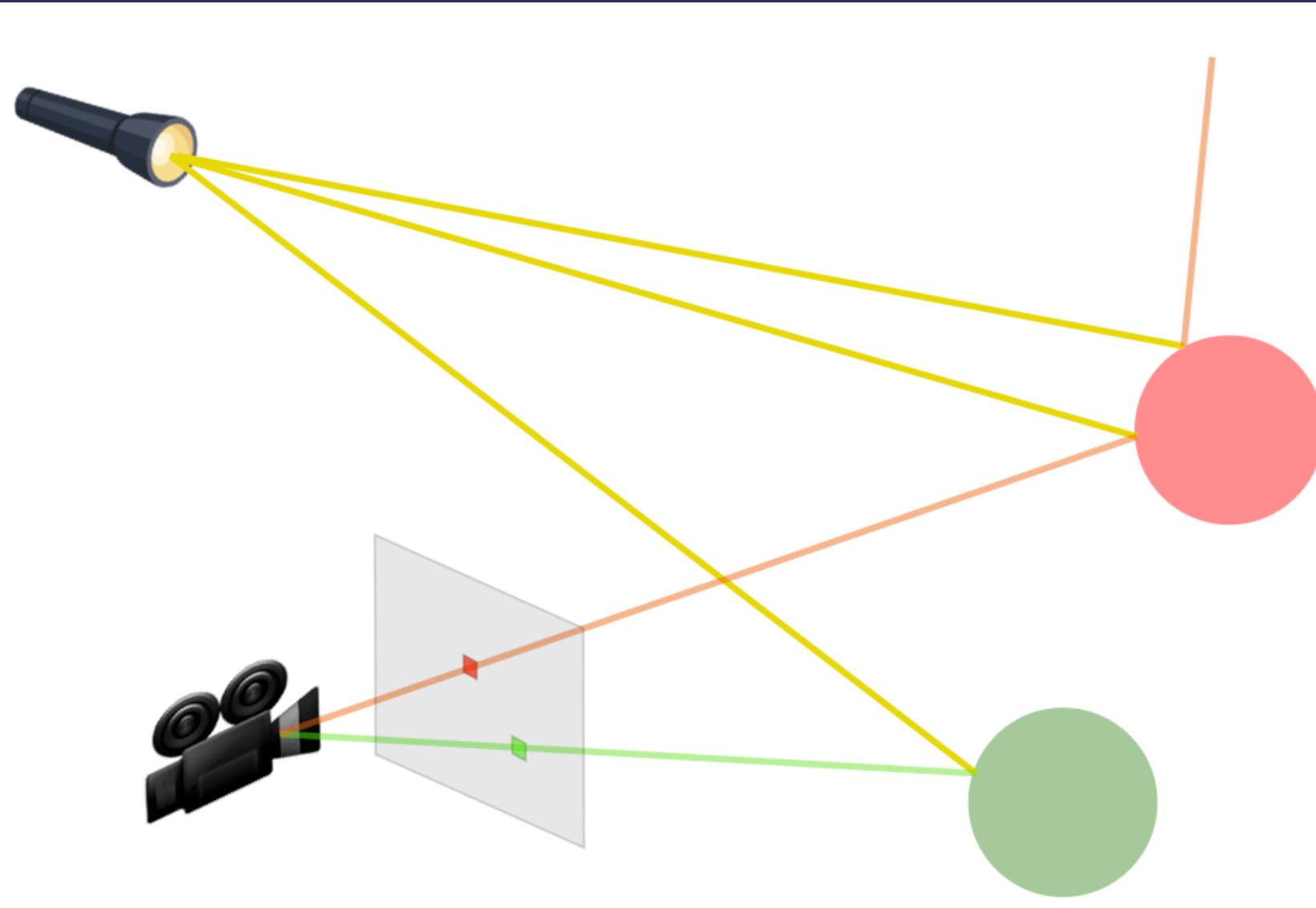
# World Rendering - Top Down



Rastering - Live

```
object CameraModule {  
    trait Service[R] {  
        def rayForPixel(  
            camera: Camera, px: Int, py: Int  
        ): ZIO[R, Nothing, Ray]  
    }  
}  
  
object WorldModule {  
    trait Service[R] {  
        def colorForRay(  
            world: World, ray: Ray  
        ): ZIO[R, RayTracerError, Color]  
    }  
}
```

# World Rendering - Top Down



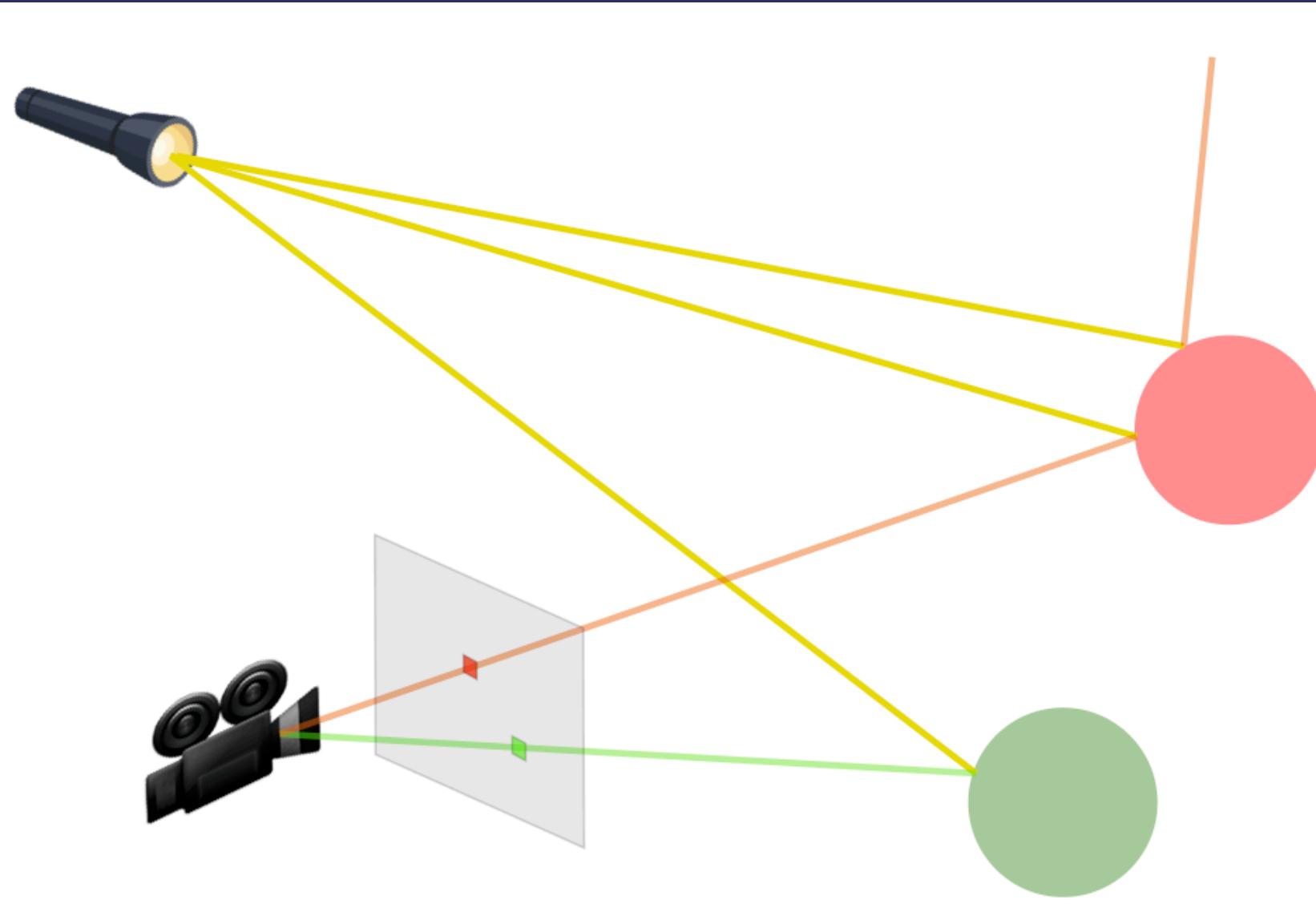
## Rastering - Live

→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



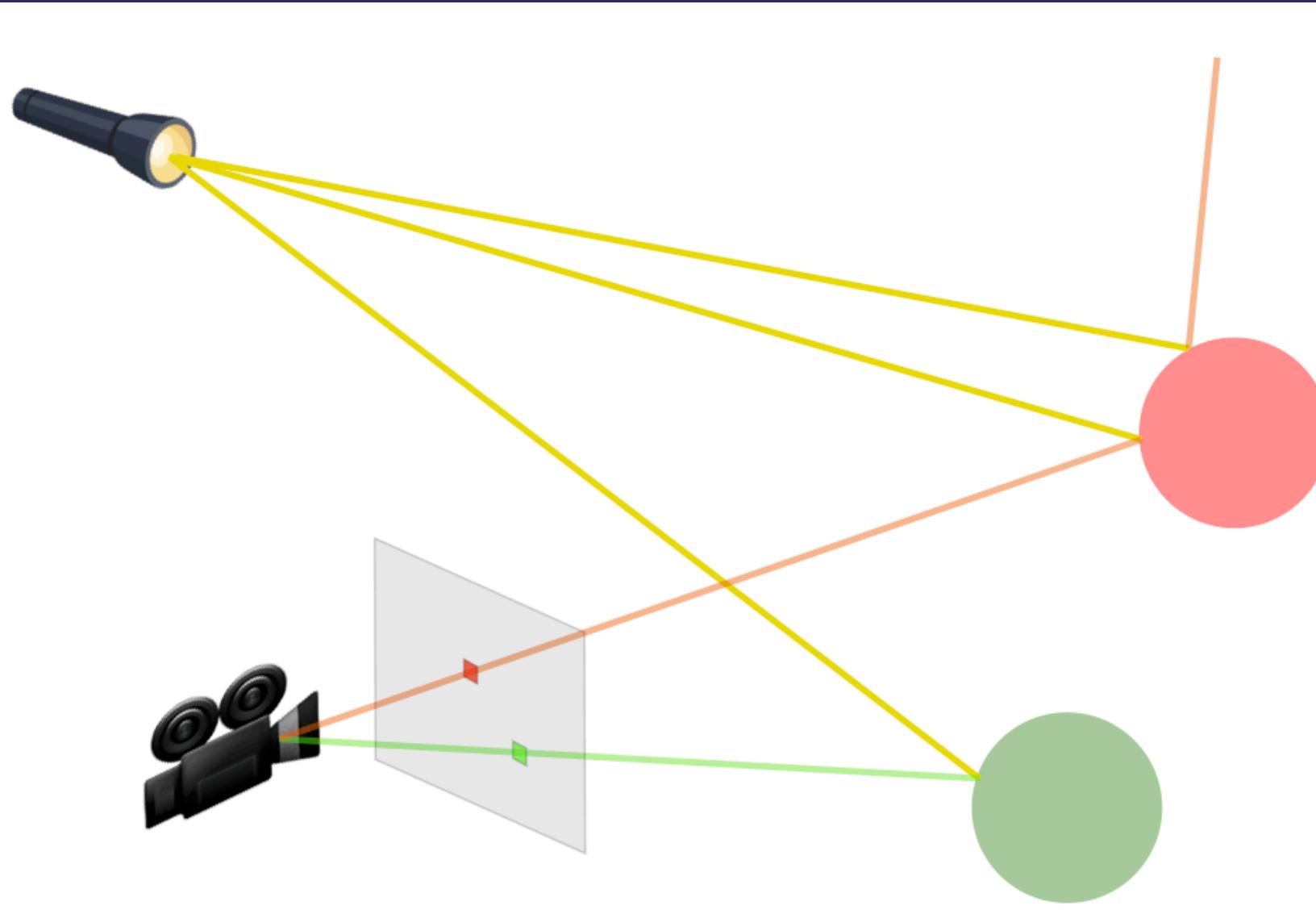
## Rastering - Live

→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



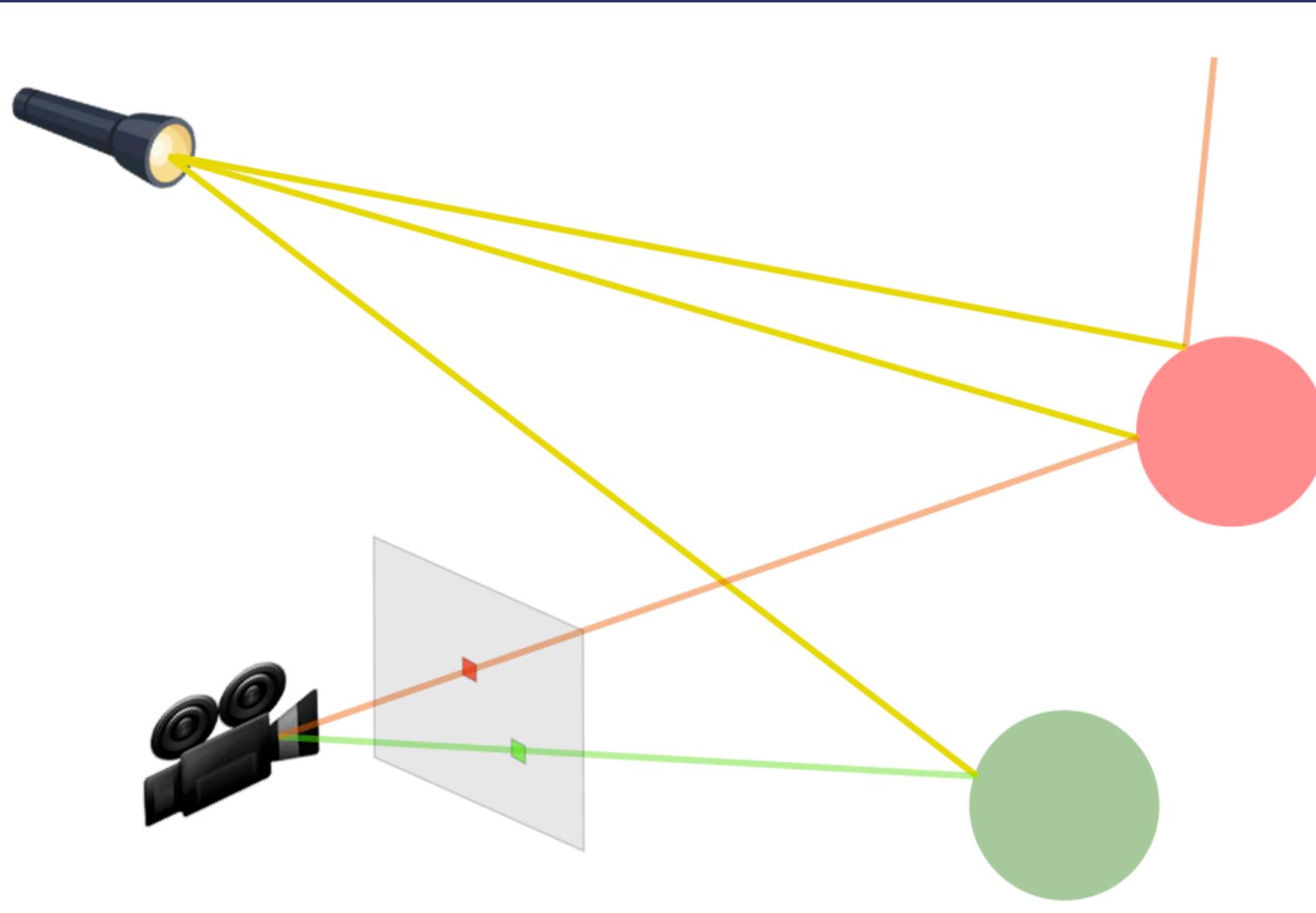
## Rastering - Live

→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



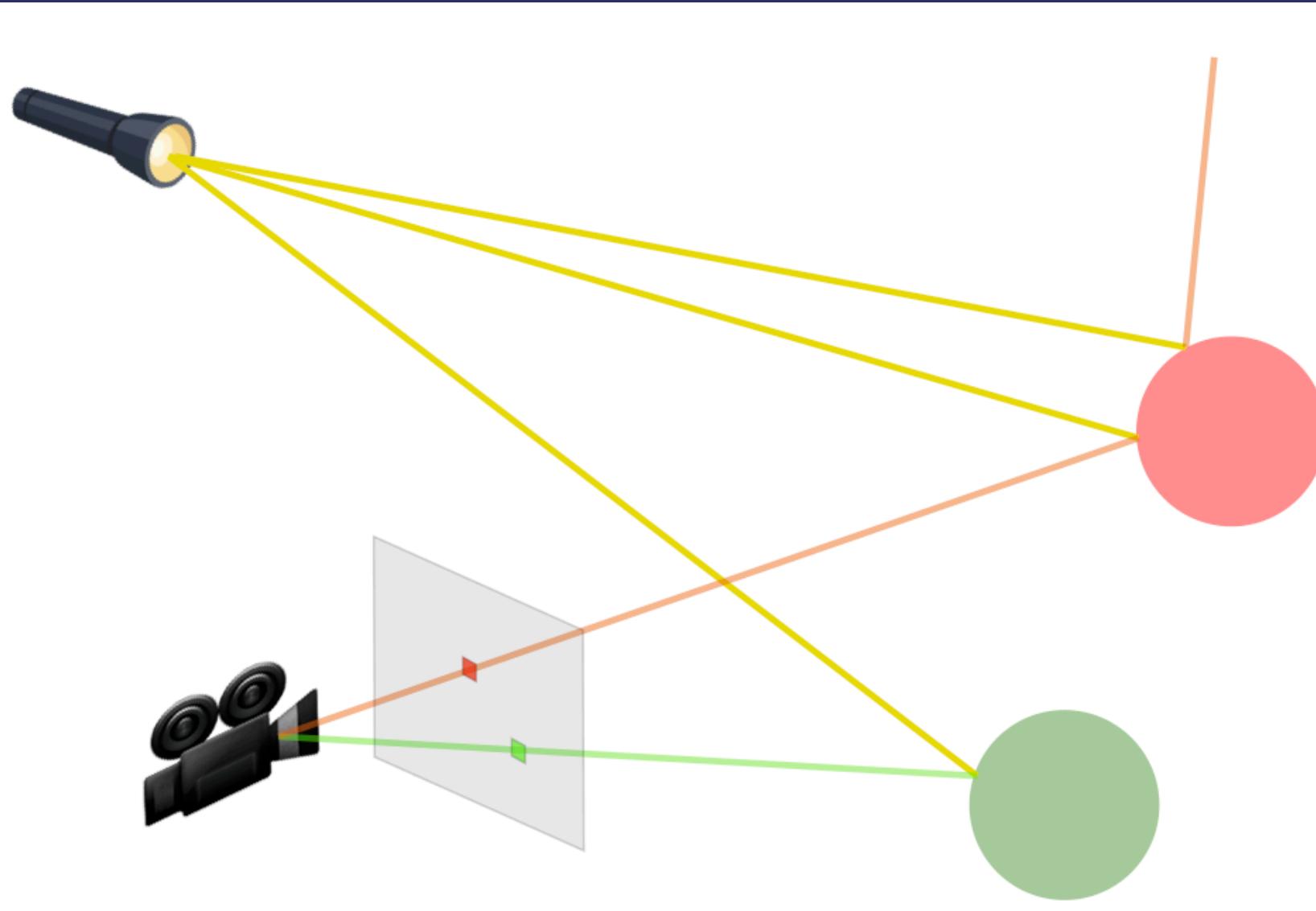
## Rastering - Live

→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



## Rastering - Live

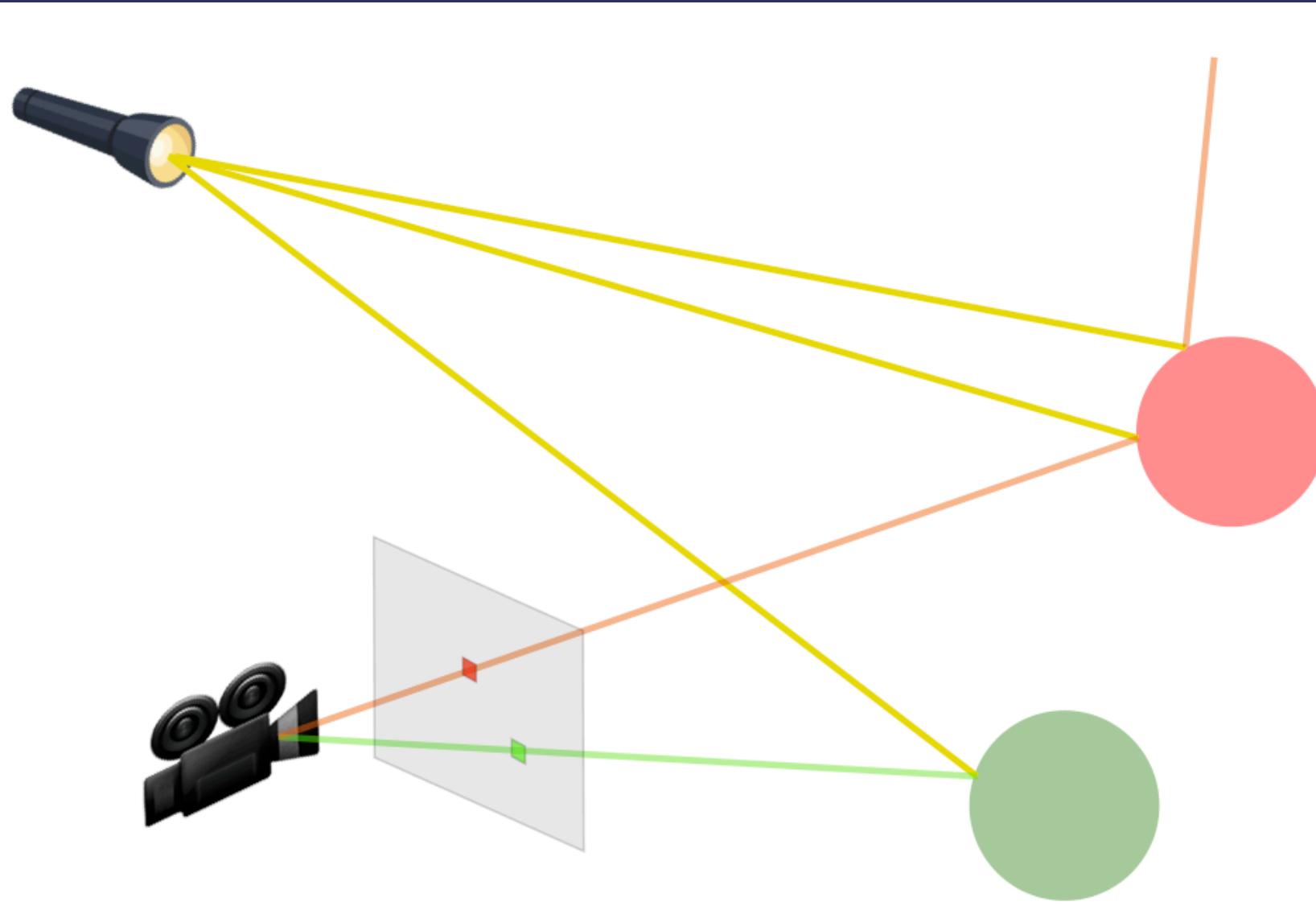
→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

→ World module - Color per ray

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



## Rastering - Live

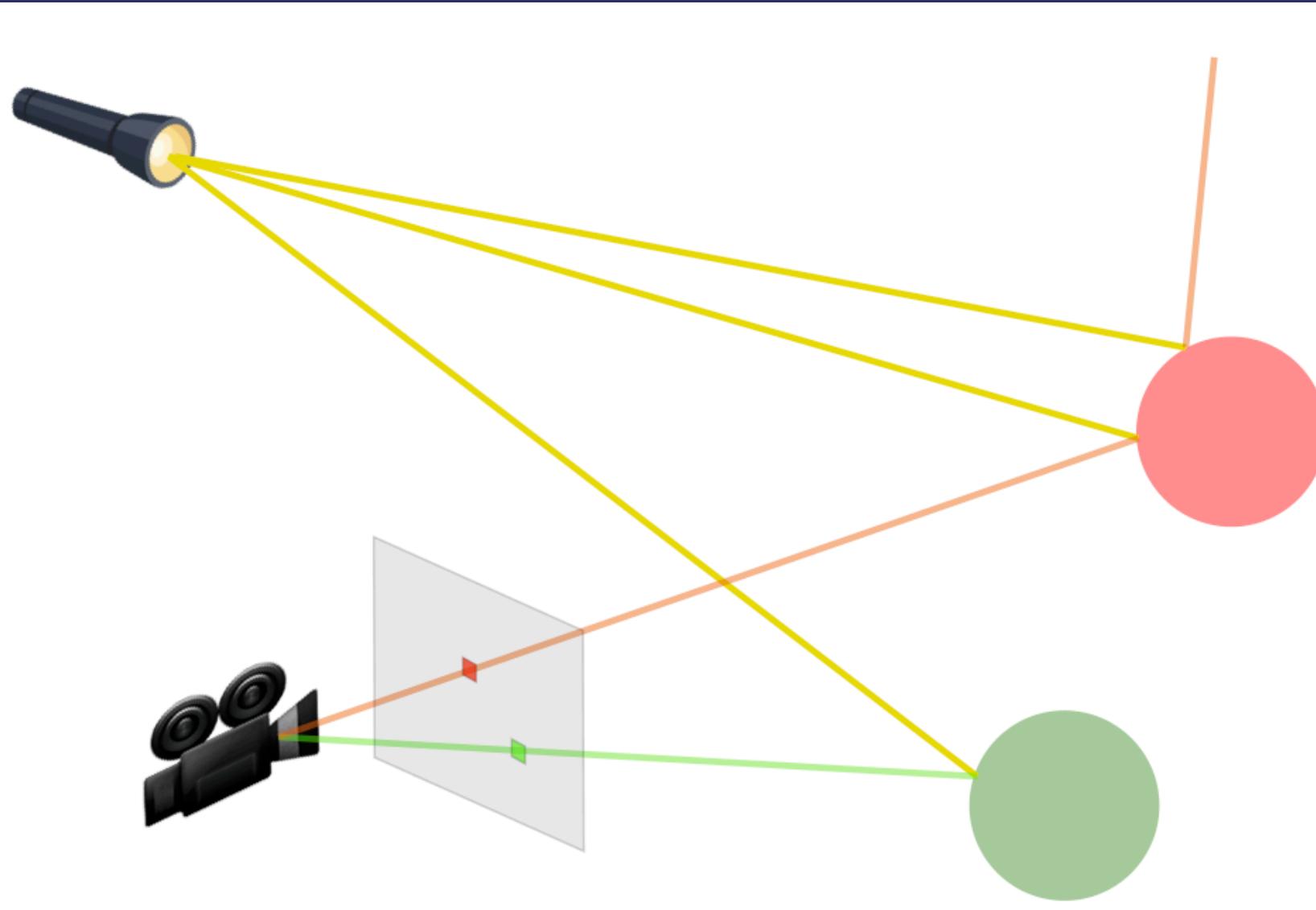
→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

→ World module - Color per ray

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down



## Rastering - Live

→ Camera module - Ray per pixel

```
object CameraModule {  
  trait Service[R] {  
    def rayForPixel(  
      camera: Camera, px: Int, py: Int  
    ): ZIO[R, Nothing, Ray]  
  }  
}
```

→ World module - Color per ray

```
object WorldModule {  
  trait Service[R] {  
    def colorForRay(  
      world: World, ray: Ray  
    ): ZIO[R, RayTracerError, Color]  
  }  
}
```

# World Rendering - Top Down

## Rasterizing **Live** - Module Dependency

```
trait LiveRasteringModule extends RasteringModule {
    val cameraModule: CameraModule.Service[Any]
    val worldModule: WorldModule.Service[Any]

    val rasteringModule: Service[Any] = new Service[Any] {
        def raster(world: World, camera: Camera): ZStream[Any, RayTracerError, ColoredPixel] = {
            val pixels: Stream[Nothing, (Int, Int)] = ???
            pixels.mapM{
                case (px, py) =>
                    for {
                        ray   <- cameraModule.rayForPixel(camera, px, py)
                        color <- worldModule.colorForRay(world, ray)
                    } yield data.ColoredPixel(Pixel(px, py), color)
            }
        }
    }
}
```

# World Rendering - Top Down

## Rasterizing Live - Module Dependency

```
trait LiveRasteringModule extends RasteringModule {
    val cameraModule: CameraModule.Service[Any]
    val worldModule: WorldModule.Service[Any]

    val rasteringModule: Service[Any] = new Service[Any] {
        def raster(world: World, camera: Camera): ZStream[Any, RayTracerError, ColoredPixel] = {
            val pixels: Stream[Nothing, (Int, Int)] = ???
            pixels.mapM{
                case (px, py) =>
                    for {
                        ray   <- cameraModule.rayForPixel(camera, px, py)
                        color <- worldModule.colorForRay(world, ray)
                    } yield data.ColoredPixel(Pixel(px, py), color)
            }
        }
    }
}
```

# World Rendering - Top Down

## Rasterizing Live - Module Dependency

```
trait LiveRasteringModule extends RasteringModule {
    val cameraModule: CameraModule.Service[Any]
    val worldModule: WorldModule.Service[Any]

    val rasteringModule: Service[Any] = new Service[Any] {
        def raster(world: World, camera: Camera): ZStream[Any, RayTracerError, ColoredPixel] = {
            val pixels: Stream[Nothing, (Int, Int)] = ???
            pixels.mapM{
                case (px, py) =>
                    for {
                        ray   <- cameraModule.rayForPixel(camera, px, py)
                        color <- worldModule.colorForRay(world, ray)
                    } yield data.ColoredPixel(Pixel(px, py), color)
            }
        }
    }
}
```

# Test **LiveRasteringModule**

1 - Define the method under test

```
val world = /* prepare a world */  
val camera = /* prepare a camera */  
  
val appUnderTest: ZIO[RasteringModule, RayTracerError, List[ColoredPixel]] =  
  RasteringModule.>.raster(world, camera)  
    .flatMap(_.runCollect)
```

## Test **LiveRasteringModule**

2 - Annotate the modules as mockable

```
import zio.macros.annotation.mockable
```

```
@mockable  
trait CameraModule { ... }
```

```
@mockable  
trait WorldModule { ... }
```

# Test LiveRasteringModule

## 3 - Build the expectations

```
val rayForPixelExp: Expectation[CameraModule, Nothing, Ray] =  
  (CameraModule(rayForPixel(equalTo((camera, 0, 0)))) returns value(r1)) *>  
  (CameraModule(rayForPixel(equalTo((camera, 0, 1)))) returns value(r2))
```

```
val colorForRayExp: Expectation[WorldModule, Nothing, Color] =  
  (WorldModule(colorForRay(equalTo((world, r1, 5)))) returns value(Color.red)) *>  
  (WorldModule(colorForRay(equalTo((world, r2, 5)))) returns value(Color.green))
```

# Test **LiveRasteringModule**

4 - Build the environment for the code under test

```
val appUnderTest: ZIO[RasteringModule, RayTracerError, List[ColoredPixel]] =  
  RasteringModule.>.raster(world, camera)  
  .flatMap(_.runCollect)  
  
appUnderTest.provideManaged(  
  worldModuleExp.managedEnv.zipWith(cameraModuleExp.managedEnv) { (wm, cm) =>  
    new LiveRasteringModule {  
      val cameraModule: CameraModule.Service[Any] = cm.cameraModule  
      val worldModule: WorldModule.Service[Any] = wm.worldModule  
    }  
  }  
)
```

# Test **LiveRasteringModule**

4 - Build the environment for the code under test

```
val appUnderTest: ZIO[RasteringModule, RayTracerError, List[ColoredPixel]] =  
  RasteringModule.>.raster(world, camera)  
  .flatMap(_.runCollect)  
  
appUnderTest.provideManaged(  
  worldModuleExp.managedEnv.zipWith(cameraModuleExp.managedEnv) { (wm, cm) =>  
    new LiveRasteringModule {  
      val cameraModule: CameraModule.Service[Any] = cm.cameraModule  
      val worldModule: WorldModule.Service[Any] = wm.worldModule  
    }  
  }  
)
```

# Test **LiveRasteringModule**

5 - Assert on the results

```
assert(res, equalTo(List(  
    ColoredPixel(Pixel(0, 0), Color.red),  
    ColoredPixel(Pixel(0, 1), Color.green),  
    ColoredPixel(Pixel(1, 0), Color.blue),  
    ColoredPixel(Pixel(1, 1), Color.white),  
)))  
)
```

# Test LiveRasteringModule

```
suite("LiveRasteringModule") {
  testM("raster should rely on cameraModule and world module") {
    for {
      (worldModuleExp, cameraModuleExp) <- RasteringModuleMocks.mockExpectations(world, camera)
      res <- appUnderTest.provideManaged(
        worldModuleExp.managedEnv.zipWith(cameraModuleExp.managedEnv) { (wm, cm) =>
          new LiveRasteringModule {
            val cameraModule: CameraModule.Service[Any] = cm.cameraModule
            val worldModule: WorldModule.Service[Any] = wm.worldModule
          }
        }
      ) yield assert(res, equalTo(List(
        ColoredPixel(Pixel(0, 0), Color.red),
        ColoredPixel(Pixel(0, 1), Color.green)
      )))
    }
  }
}
```

# Test

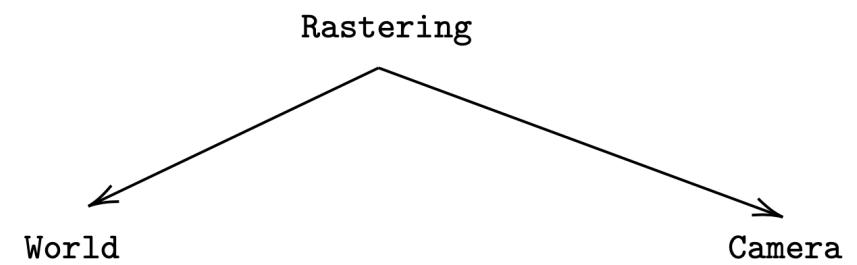
**Takeaway: Implement and test every layer only in terms of the immediately underlying layer**

**IT'S MODULES**



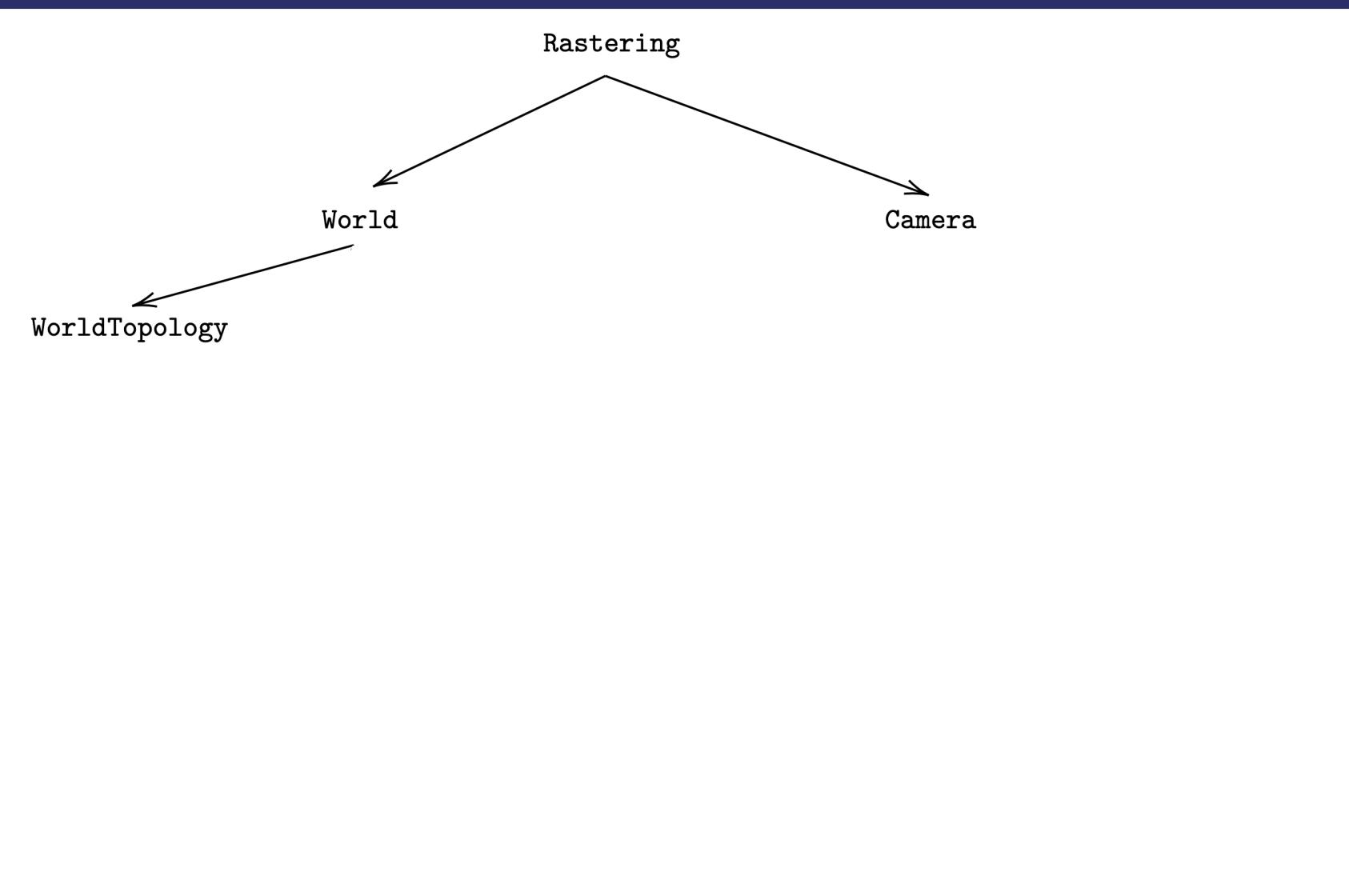
**ALL THE WAY DOWN**

# Live CameraModule



```
trait Live extends CameraModule {  
    val aTModule: ATModule.Service[Any]  
    /* implementation */  
}
```

# Live WorldModule



## WorldTopologyModule

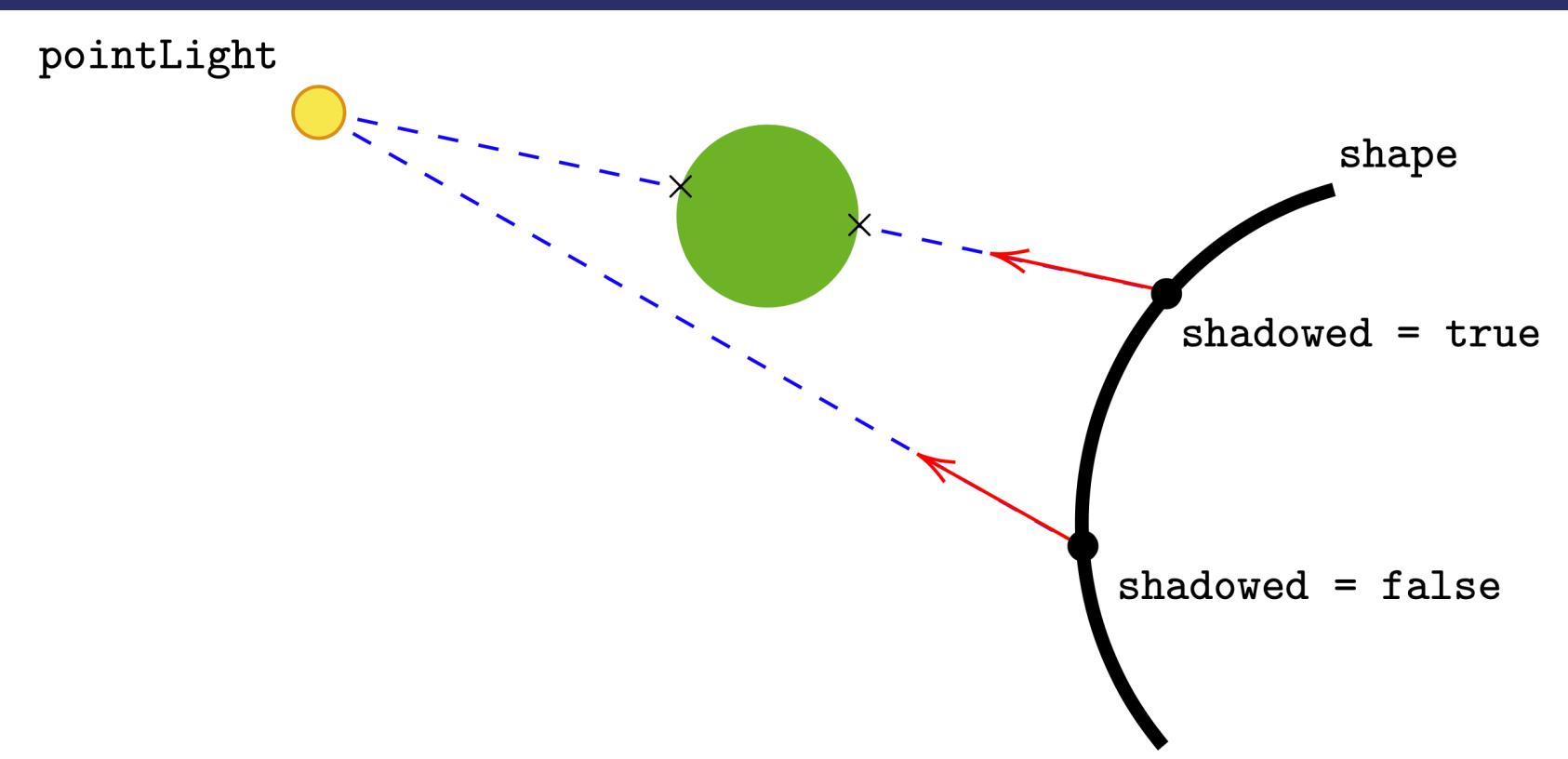
```
trait Live extends WorldModule {
    val worldTopologyModule: WorldTopologyModule.Service[Any]

    val worldModule: Service[Any] = new Service[Any] {
        def colorForRay(
            world: World, ray: Ray, remaining: Ref[Int]
        ): ZIO[Any, RayTracerError, Color] = {
            /* use other modules */
        }
    }
}
```

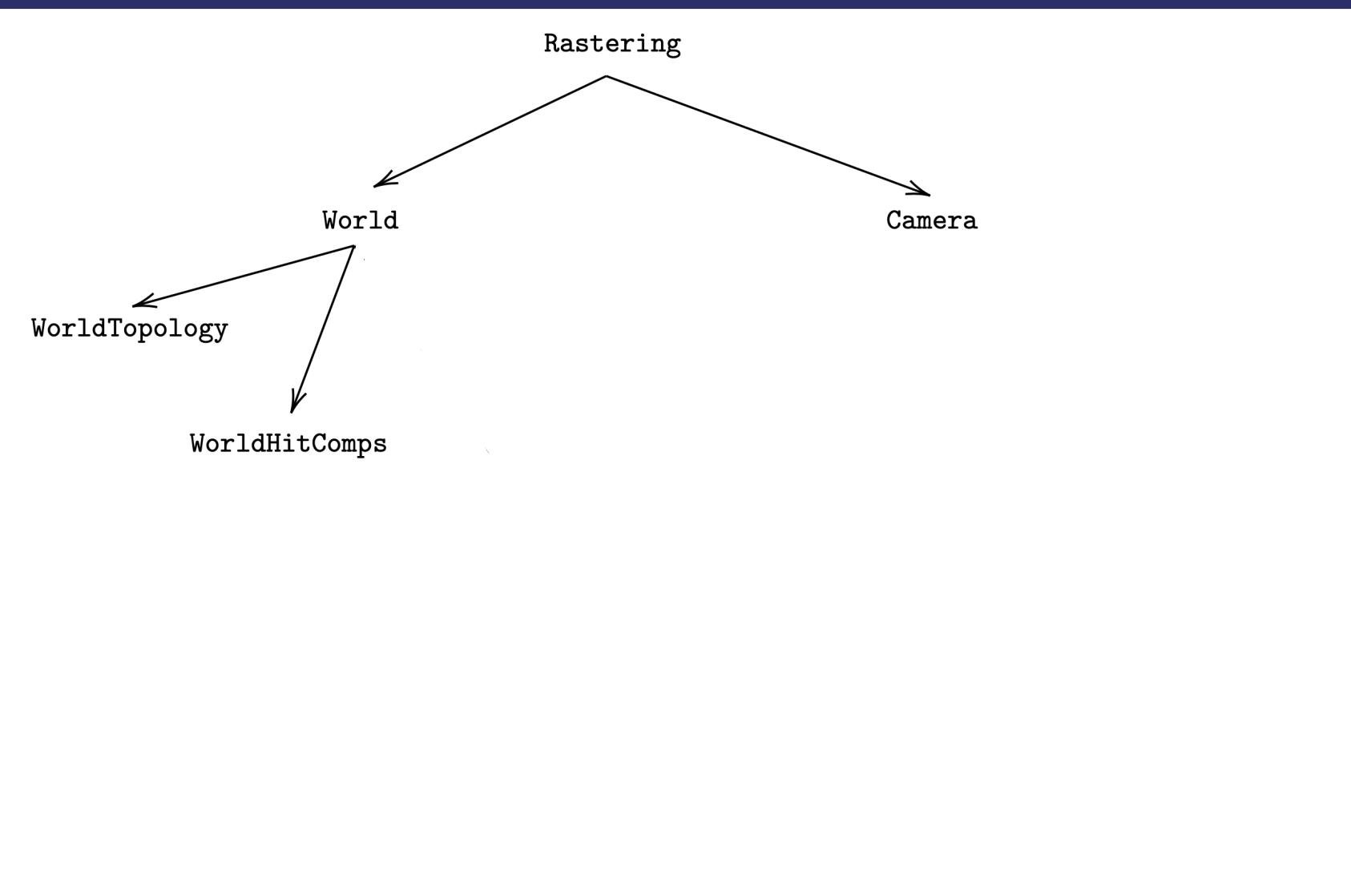
# Live WorldModule

## WorldTopologyModule

```
trait Live extends WorldModule {  
    val worldTopologyModule: WorldTopologyModule.Service[Any]  
  
    val worldModule: Service[Any] = new Service[Any] {  
        def colorForRay(  
            world: World, ray: Ray, remaining: Ref[Int]  
        ): ZIO[Any, RayTracerError, Color] = {  
            /* use other modules */  
        }  
    }  
}
```



# Live WorldModule



## WorldHitCompsModule

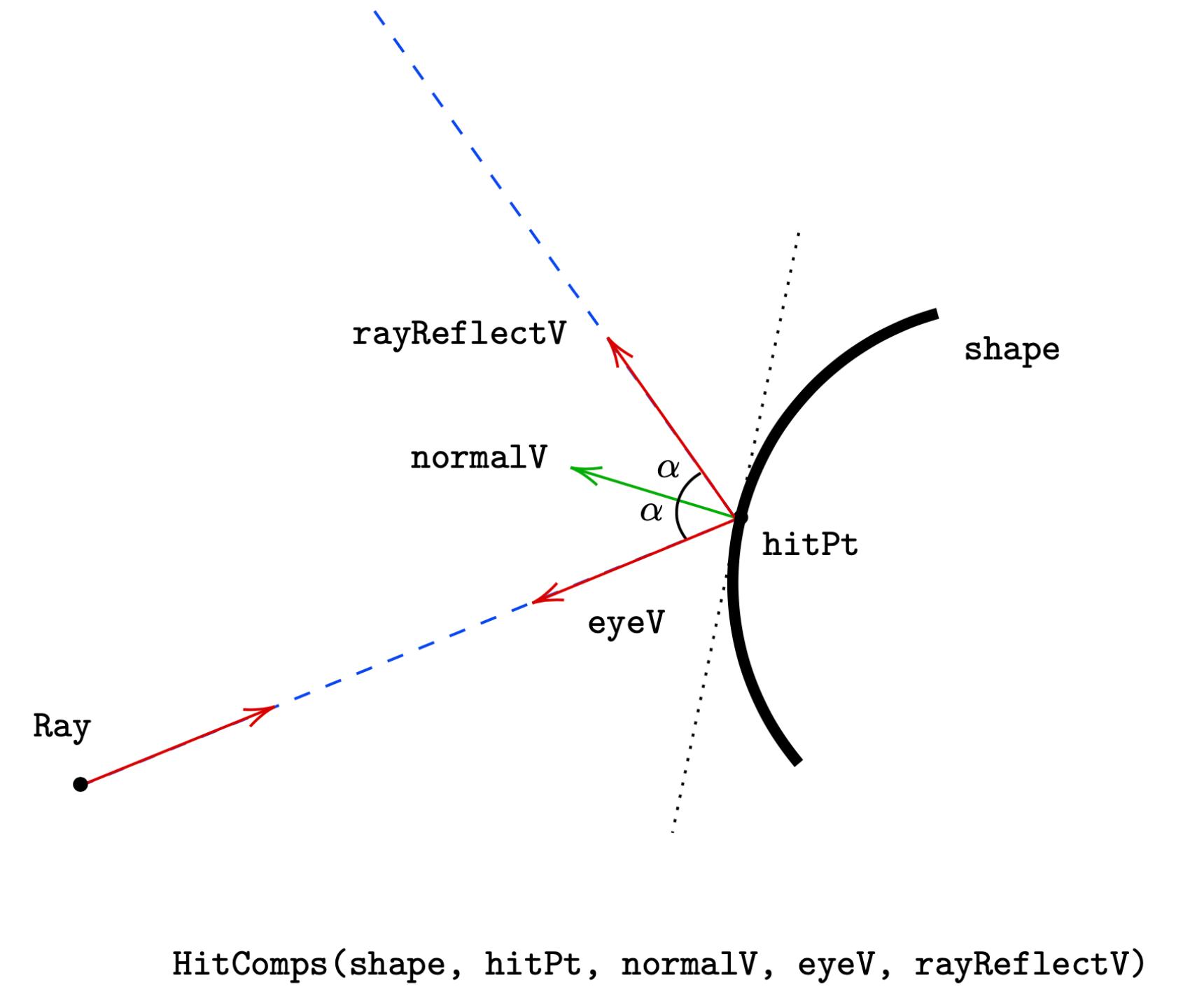
```
trait Live extends WorldModule {
    val worldTopologyModule: WorldTopologyModule.Service[Any]
    val worldHitCompsModule: WorldHitCompsModule.Service[Any]

    val worldModule: Service[Any] = new Service[Any] {
        def colorForRay(
            world: World, ray: Ray, remaining: Ref[Int]
        ): ZIO[Any, RayTracerError, Color] = {
            /* use other modules */
        }
    }
}
```

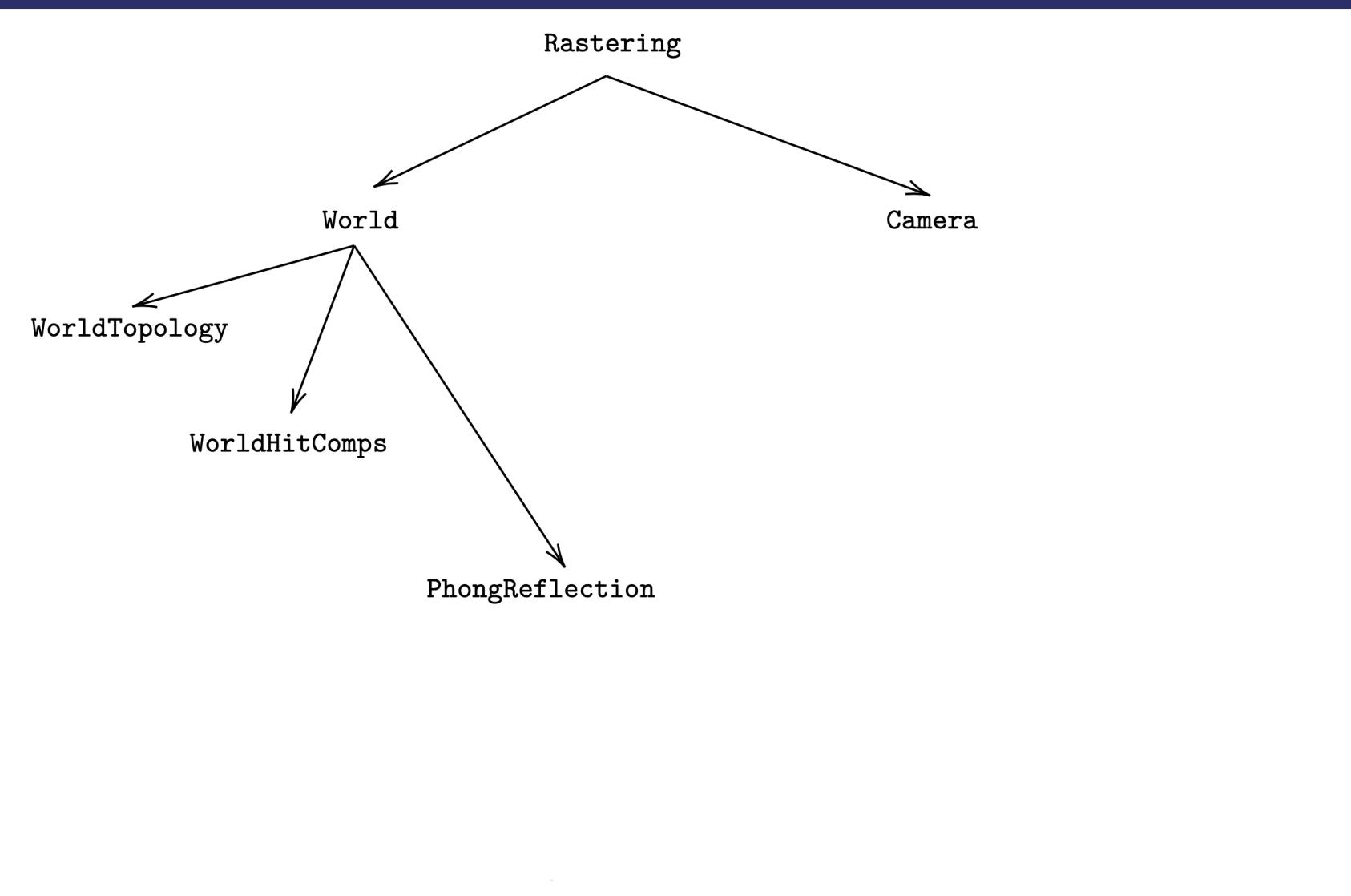
# Live WorldModule

## WorldHitCompsModule

```
case class HitComps(  
  shape: Shape, hitPt: Pt, normalV: Vec, eyeV: Vec, rayReflectV: Vec  
)  
  
trait Service[R] {  
  def hitComps(  
    ray: Ray, hit: Intersection, intersections: List[Intersection]  
  ): ZIO[R, Nothing, HitComps]  
}
```



# Live WorldModule



## PhongReflectionModule

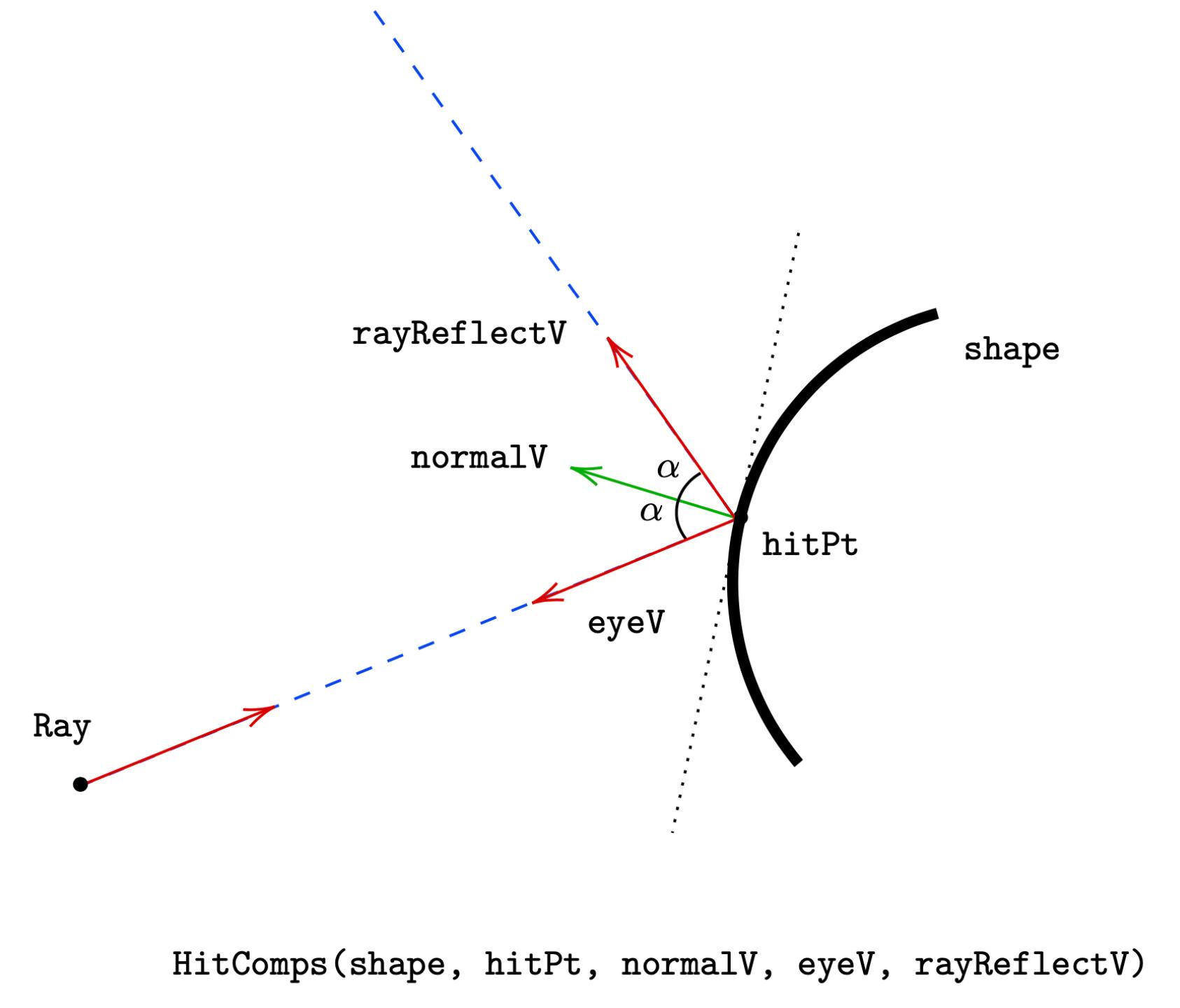
```
trait Live extends WorldModule {
    val worldTopologyModule: WorldTopologyModule.Service[Any]
    val worldHitCompsModule: WorldHitCompsModule.Service[Any]
    val phongReflectionModule: PhongReflectionModule.Service[Any]

    val worldModule: Service[Any] = new Service[Any] {
        def colorForRay(
            world: World, ray: Ray, remaining: Ref[Int]
        ): ZIO[Any, RayTracerError, Color] = {
            /* use other modules */
        }
    }
}
```

# Live WorldModule

## PhongReflectionModule

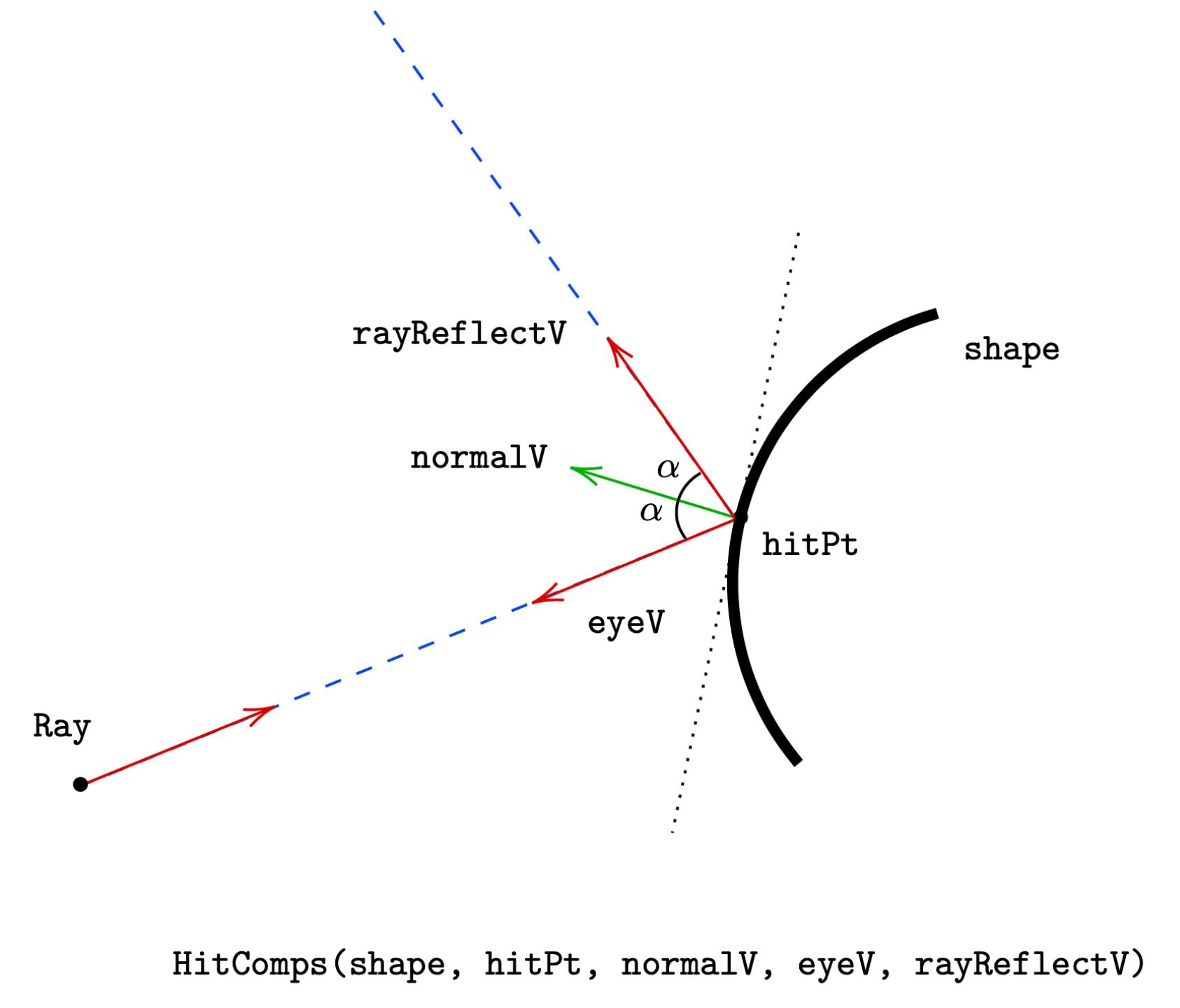
```
case class PhongComponents(  
    ambient: Color, diffuse: Color, reflective: Color  
) {  
    def toColor: Color = ambient + diffuse + reflective  
}  
  
trait BlackWhite extends PhongReflectionModule {  
    val phongReflectionModule: Service[Any] =  
        new Service[Any] {  
            def lighting(  
                pointLight: PointLight, hitComps: HitComps, inShadow: Boolean  
) : UIO[PhongComponents] = {  
                if (inShadow) UIO(PhongComponents.allBlack)  
                else UIO(PhongComponents.allWhite)  
            }  
        }  
}
```



# Live WorldModule

## PhongReflectionModule

```
case class PhongComponents(  
    ambient: Color, diffuse: Color, reflective: Color  
) {  
    def toColor: Color = ambient + diffuse + reflective  
}  
  
trait BlackWhite extends PhongReflectionModule {  
    val phongReflectionModule: Service[Any] =  
        new Service[Any] {  
            def lighting(  
                pointLight: PointLight, hitComps: HitComps, inShadow: Boolean  
) : UIO[PhongComponents] = {  
                if (inShadow) UIO(PhongComponents.allBlack)  
                else UIO(PhongComponents.allWhite)  
            }  
        }  
}
```



# Display the first canvas / 1

```
def drawOnCanvasWithCamera(world: World, camera: Camera, canvas: Canvas) =  
  for {  
    coloredPointsStream <- RasteringModule.>.raster(world, camera)  
    _ <- coloredPointsStream.mapM(cp => canvas.update(cp)).run(Sink.drain)  
  } yield ()  
  
def program(viewFrom: Pt) =  
  for {  
    camera <- cameraFor(viewFrom: Pt)  
    w      <- world  
    canvas <- Canvas.create()  
    _      <- drawOnCanvasWithCamera(w, camera, canvas)  
    _      <- CanvasSerializer.>.serialize(canvas, 255)  
  } yield ()
```

# Display the first canvas / 2

```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasteringModule with ATModule, RayTracerError, Unit]  
  
program(Pt(2, 2, -10))  
.provide(  
  new CanvasSerializer.PNGCanvasSerializer  
  with RasteringModule.ChunkRasteringModule  
  with ATModule.Live  
)  
// Members declared in zio.blocking.Blocking  
// [error]  val blocking: zio.blocking.Blocking.Service[Any] = ???  
// [error]  
// [error]  // Members declared in modules.RasteringModule.ChunkRasteringModule  
// [error]  val cameraModule: modules.CameraModule.Service[Any] = ???  
// [error]  val worldModule: modules.WorldModule.Service[Any] = ???  
// [error]  
// [error]  // Members declared in geometry.affine.ATModule.Live  
// [error]  val matrixModule: geometry.matrix.MatrixModule.Service[Any] = ???
```

# Display the first canvas / 2

```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasteringModule with ATModule, RayTracerError, Unit]  
  
program(Pt(2, 2, -10))  
.provide(  
  new CanvasSerializer.PNGCanvasSerializer  
  with RasteringModule.ChunkRasteringModule  
  with ATModule.Live  
)  
// Members declared in zio.blocking.Blocking  
// [error]  val blocking: zio.blocking.Blocking.Service[Any] = ???  
// [error]  
// [error]  // Members declared in modules.RasteringModule.ChunkRasteringModule  
// [error]  val cameraModule: modules.CameraModule.Service[Any] = ???  
// [error]  val worldModule: modules.WorldModule.Service[Any] = ???  
// [error]  
// [error]  // Members declared in geometry.affine.ATModule.Live  
// [error]  val matrixModule: geometry.matrix.MatrixModule.Service[Any] = ???
```

# Display the first canvas / 2

```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasteringModule with ATModule, RayTracerError, Unit]  
  
program(Pt(2, 2, -10))  
.provide(  
  new CanvasSerializer.PNGCanvasSerializer  
  with RasteringModule.ChunkRasteringModule  
  with ATModule.Live  
)  
// Members declared in zio.blocking.Blocking  
// [error]  val blocking: zio.blocking.Blocking.Service[Any] = ???  
// [error]  
// [error]  // Members declared in modules.RasteringModule.ChunkRasteringModule  
// [error]  val cameraModule: modules.CameraModule.Service[Any] = ???  
// [error]  val worldModule: modules.WorldModule.Service[Any] = ???  
// [error]  
// [error]  // Members declared in geometry.affine.ATModule.Live  
// [error]  val matrixModule: geometry.matrix.MatrixModule.Service[Any] = ???
```

# Display the first canvas / 3

```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasteringModule with ATModule, RayTracerError, Unit]  
  
program(Pt(2, 2, -10))  
  .provide(  
    new CanvasSerializer.PNGCanvasSerializer  
    with RasteringModule.ChunkRasteringModule  
    with ATModule.Live  
    with CameraModule.Live  
    with MatrixModule.BreezeLive  
    with WorldModule.Live  
  )  
)  
// [error] // Members declared in io.tuliplogic.raytracer.ops.model.modules.WorldModule.Live  
// [error] val phongReflectionModule: io.tuliplogic.raytracer.ops.model.modules.PhongReflectionModule.Service[Any] = ???  
// [error] val worldHitCompsModule: io.tuliplogic.raytracer.ops.model.modules.WorldHitCompsModule.Service[Any] = ???  
// [error] val worldReflectionModule: io.tuliplogic.raytracer.ops.model.modules.WorldReflectionModule.Service[Any] = ???  
// [error] val worldRefractionModule: io.tuliplogic.raytracer.ops.model.modules.WorldRefractionModule.Service[Any] = ???  
// [error] val worldTopologyModule: io.tuliplogic.raytracer.ops.model.modules.WorldTopologyModule.Service[Any] = ???
```

# Display the first canvas - /4

Group modules in **trait**

```
type BasicModules =  
    NormalReflectModule.Live  
    with RayModule.Live  
    with ATModule.Live  
    with MatrixModule.BreezeLive  
    with WorldModule.Live  
    with WorldTopologyModule.Live  
    with WorldHitCompsModule.Live  
    with CameraModule.Live  
    with RasteringModule.Live  
    with Blocking.Live
```

# Display the first canvas - /4

Group modules in **trait**

```
type BasicModules =  
    NormalReflectModule.Live  
    with RayModule.Live  
    with ATModule.Live  
    with MatrixModule.BreezeLive  
    with WorldModule.Live  
    with WorldTopologyModule.Live  
    with WorldHitCompsModule.Live  
    with CameraModule.Live  
    with RasteringModule.Live  
    with Blocking.Live
```

# Display the first canvas - /5

## Group modules

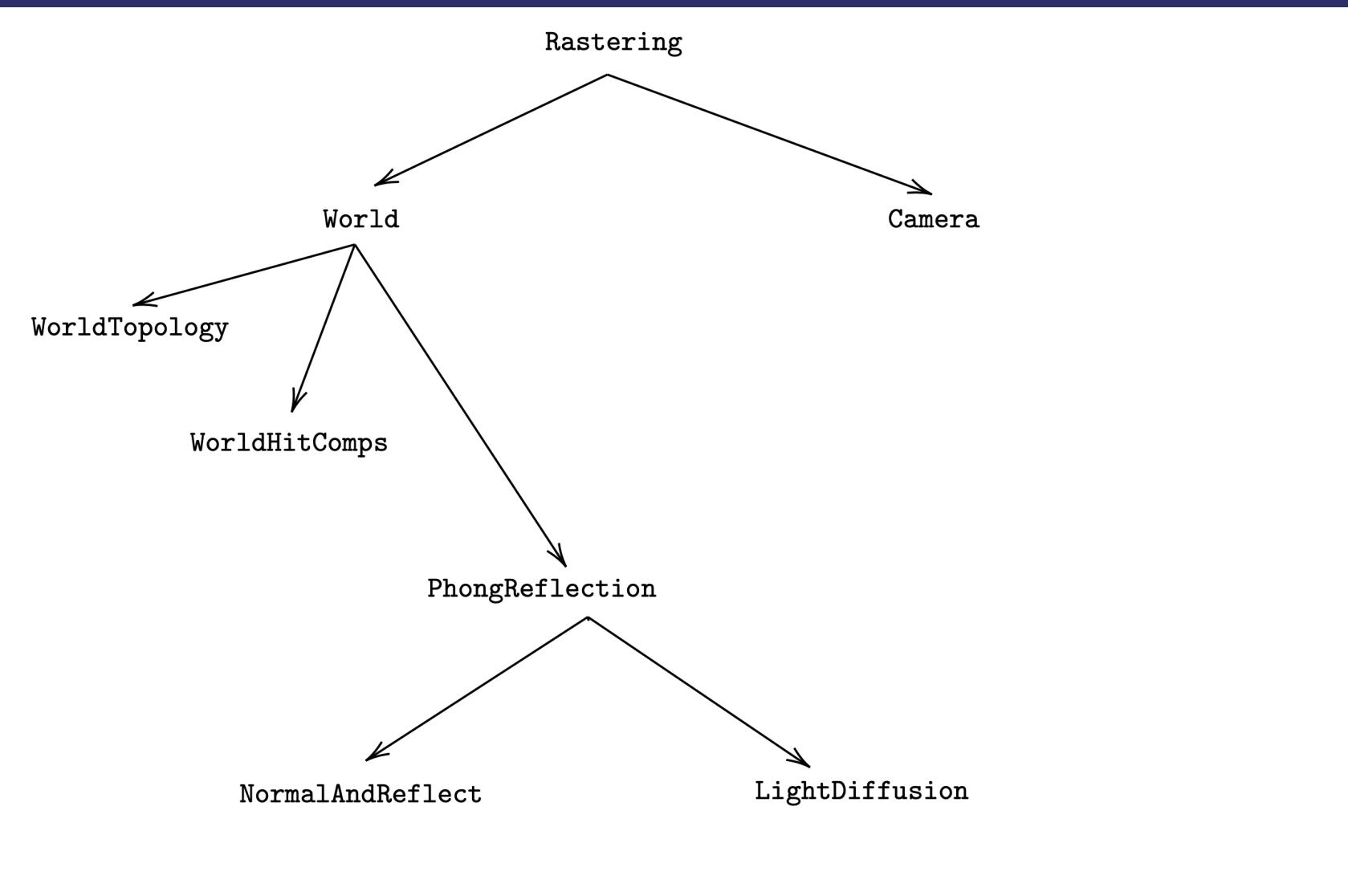
```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasteringModule with ATModule, RayTracerError, Unit]  
  
program(Pt(2, 2, -10))  
.provide(new BasicModules with PhongReflectionModule.BlackWhite)
```

# Display the first canvas - /6

## Group modules

```
def program(viewFrom: Pt):  
  ZIO[CanvasSerializer with RasterizingModule with ATModule, RayTracerError, Unit]  
  
def run(args: List[String]): ZIO[ZEnv, Nothing, Int] =  
  ZIO.traverse(-18 to -6)(z => program(Pt(2, 2, z))  
    .provide(  
      new BasicModules with PhongReflectionModule.BlackWhite  
    )  
  ).foldM(err =>  
    console.putStrLn(s"Execution failed with: $err").as(1),  
    _ => UIO.succeed(0)  
  )
```

# Live PhongReflectionModule

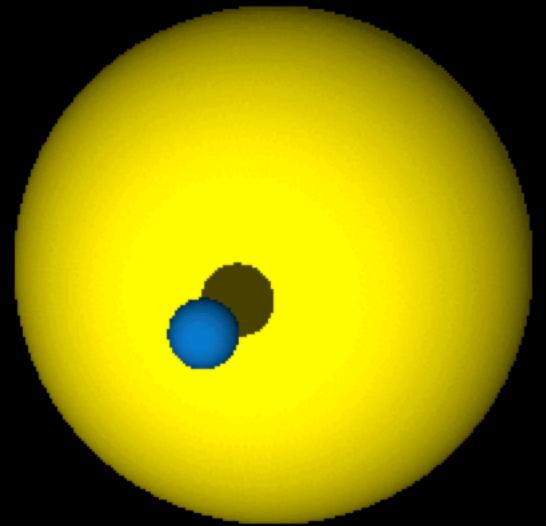


## With LightDiffusion

```
trait Live extends PhongReflectionModule {  
    val aTModule: ATModule.Service[Any]  
    val normalReflectModule: NormalReflectModule.Service[Any]  
    val lightDiffusionModule: LightDiffusionModule.Service[Any]  
}
```

# Live PhongReflectionModule

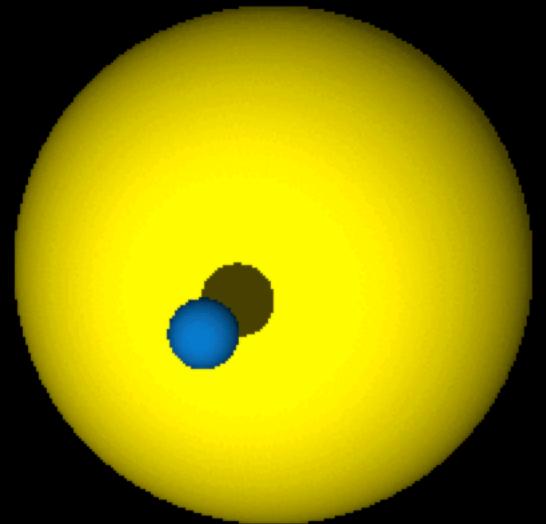
With LightDiffusion



```
program(Pt(2, 2, -10))  
.provide(  
    new BasicModules  
    with PhongReflectionModule.Live  
    // with PhongReflectionModule.BlackWhite  
)
```

# Live PhongReflectionModule

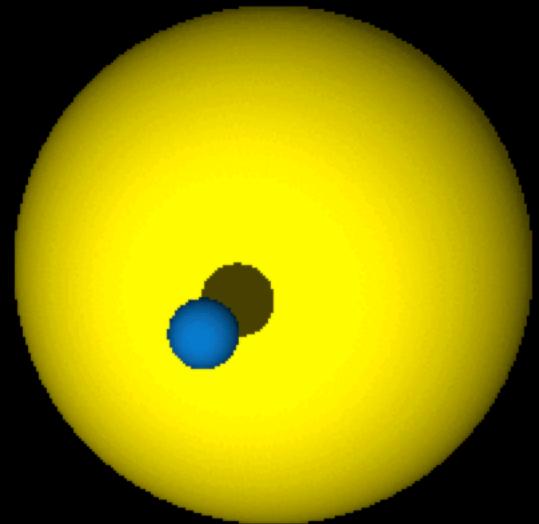
With LightDiffusion



```
program(Pt(2, 2, -10))  
.provide(  
    new BasicModules  
    with PhongReflectionModule.Live  
    // with PhongReflectionModule.BlackWhite  
)
```

# Live PhongReflectionModule

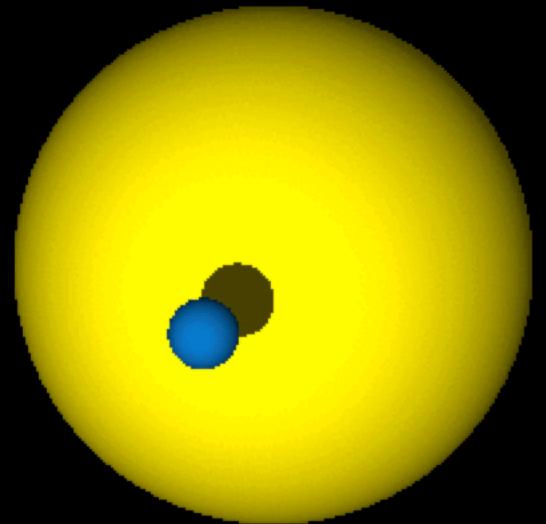
With LightDiffusion



```
program(Pt(2, 2, -10))  
.provide(  
    new BasicModules  
    with PhongReflectionModule.Live  
    // with PhongReflectionModule.BlackWhite  
)
```

# Live PhongReflectionModule

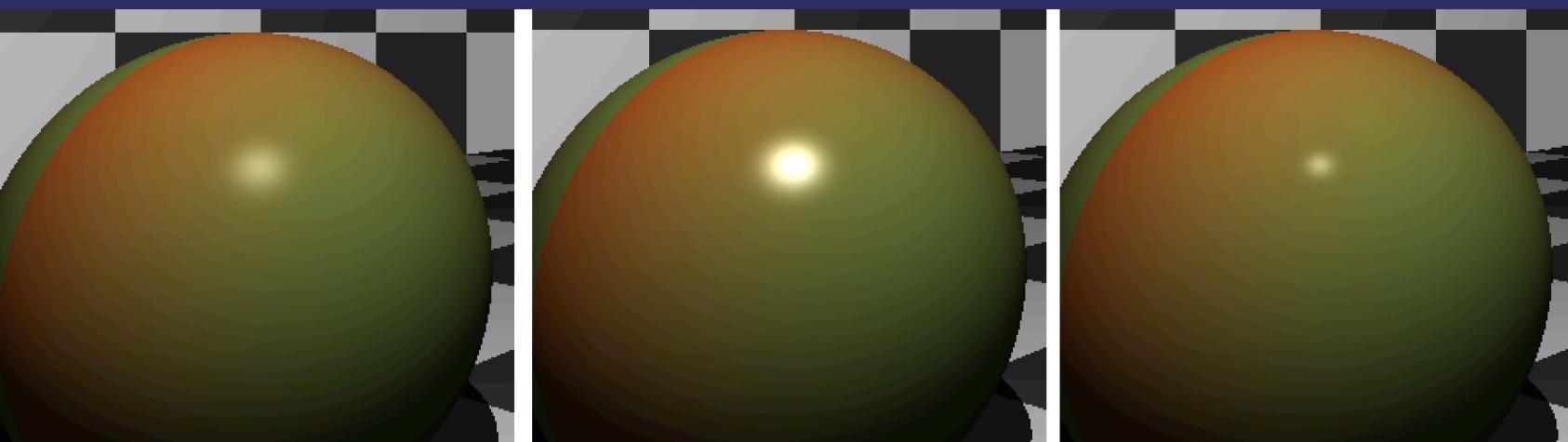
With LightDiffusion



```
program(Pt(2, 2, -10))  
.provide(  
    new BasicModules  
    with PhongReflectionModule.Live  
    // with PhongReflectionModule.BlackWhite  
)
```

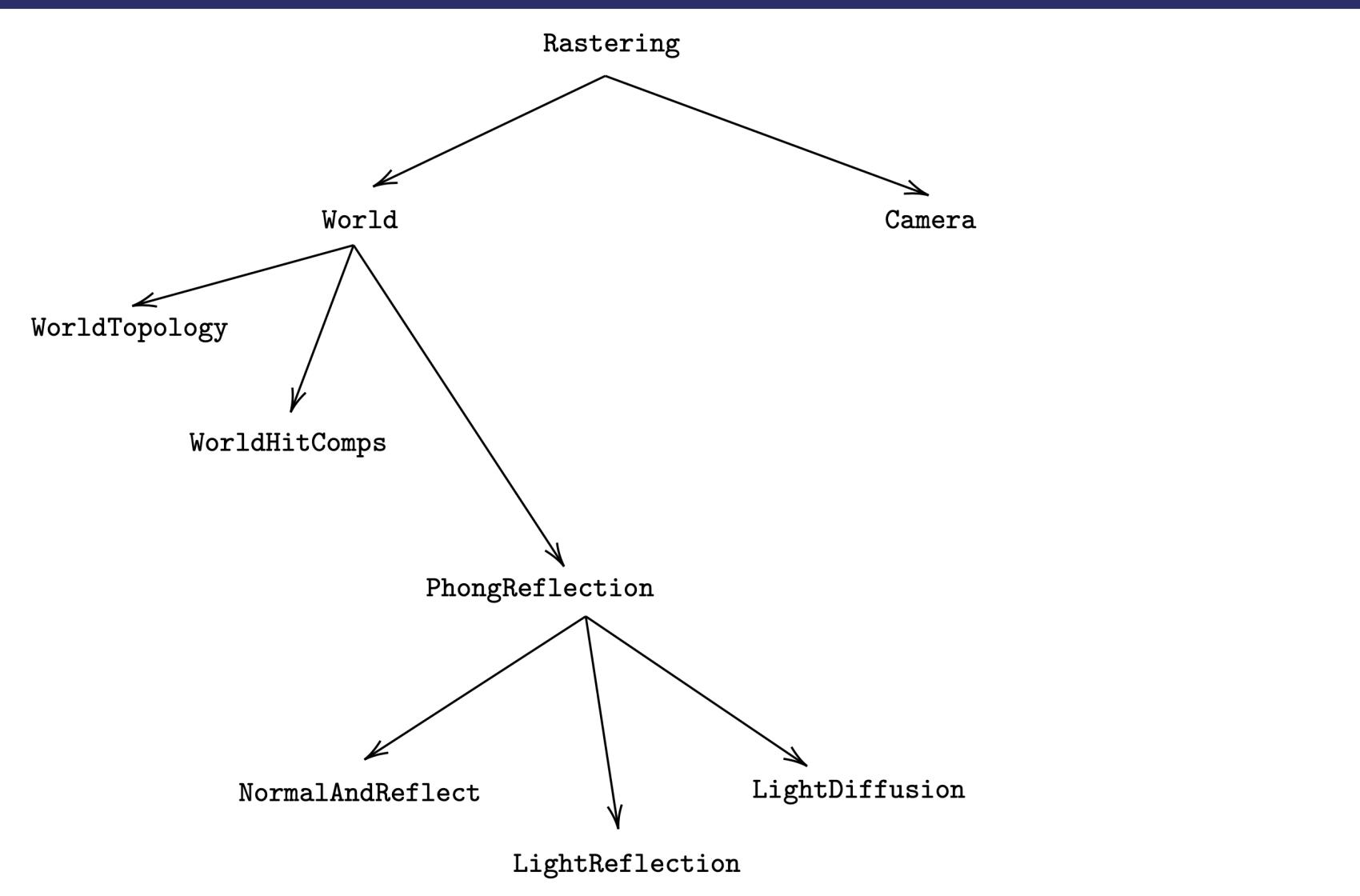
# Reflect the light source

Describe material properties



```
case class Material(  
    color: Color, // the basic color  
    ambient: Double, // ∈ [0, 1]  
    diffuse: Double, // ∈ [0, 1]  
    specular: Double, // ∈ [0, 1]  
    shininess: Double, // ∈ [10, 200]  
)
```

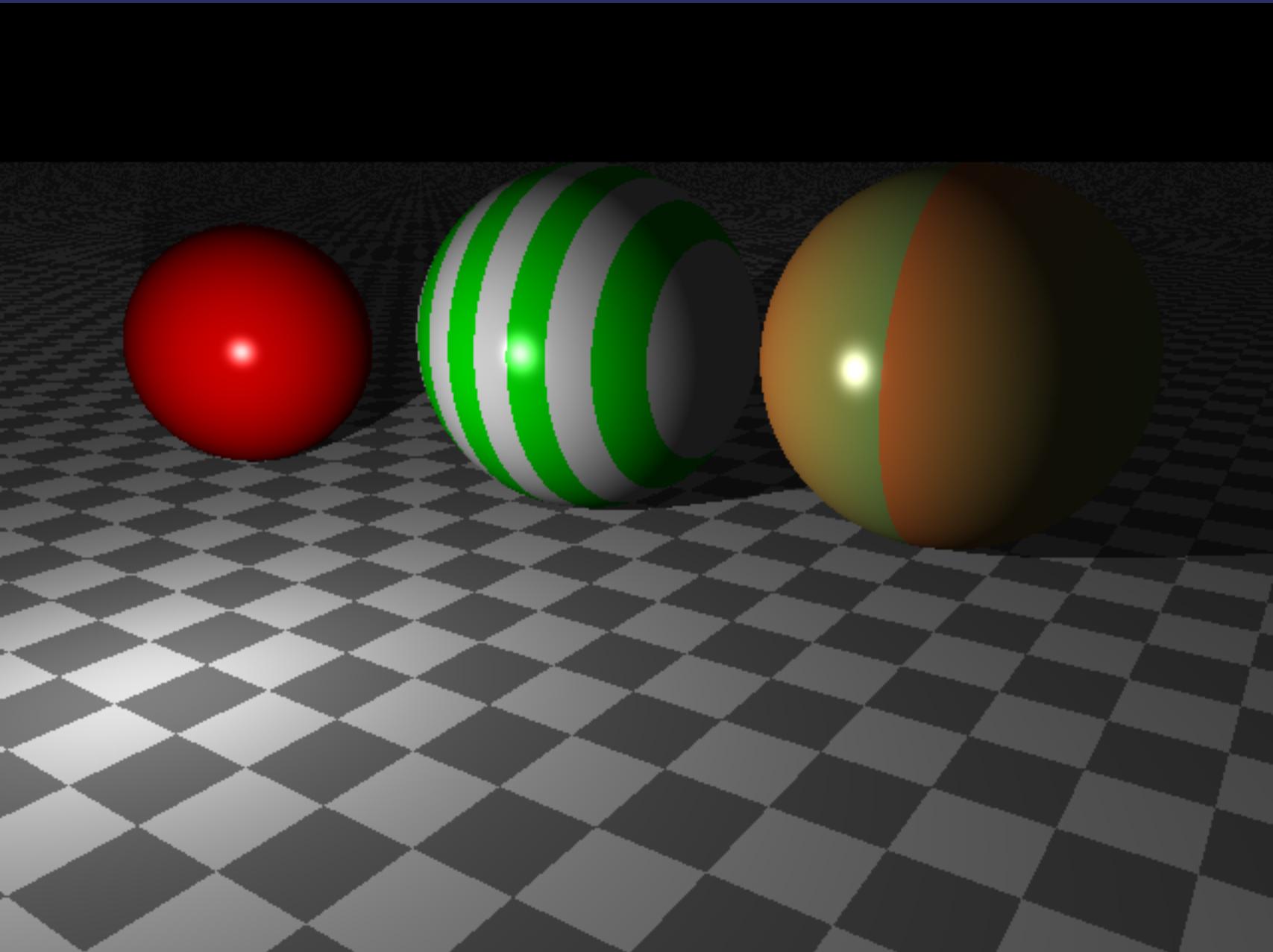
# Live PhongReflectionModule



**With LightDiffusion and LightReflection**

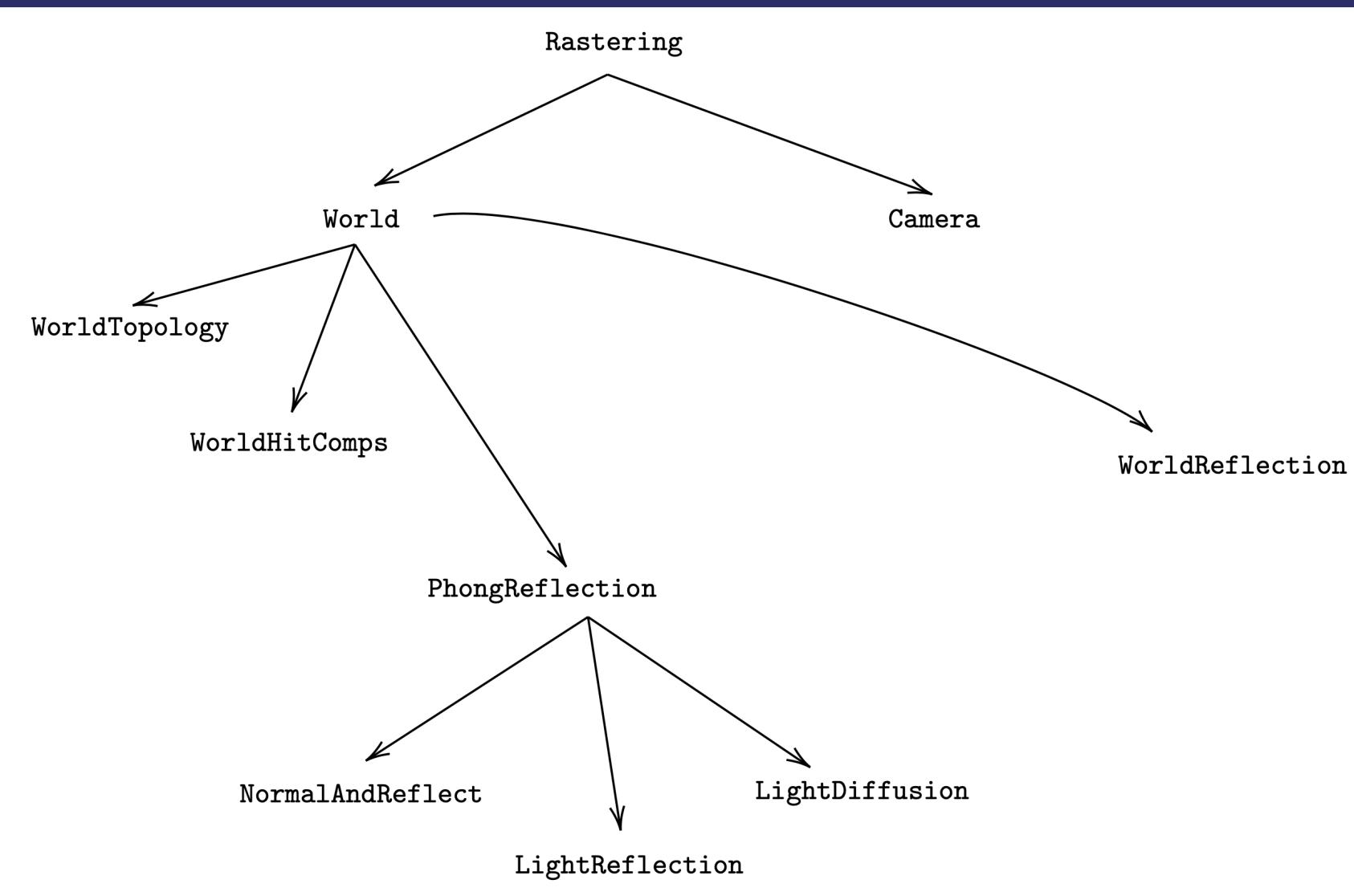
```
trait Live extends PhongReflectionModule {  
    val aTModule: ATModule.Service[Any]  
    val normalReflectModule: NormalReflectModule.Service[Any]  
    val lightDiffusionModule: LightDiffusionModule.Service[Any]  
    val lightReflectionModule: LightReflectionModule.Service[Any]  
}
```

# Spheres reflect light source



```
program(  
    from = Pt(57, 20, z),  
    to = Pt(20, 0, 20)  
).provide {  
    new BasicModules  
    with PhongReflectionModule.Live  
    with LightReflectionModule.Live  
}
```

# Reflective surfaces



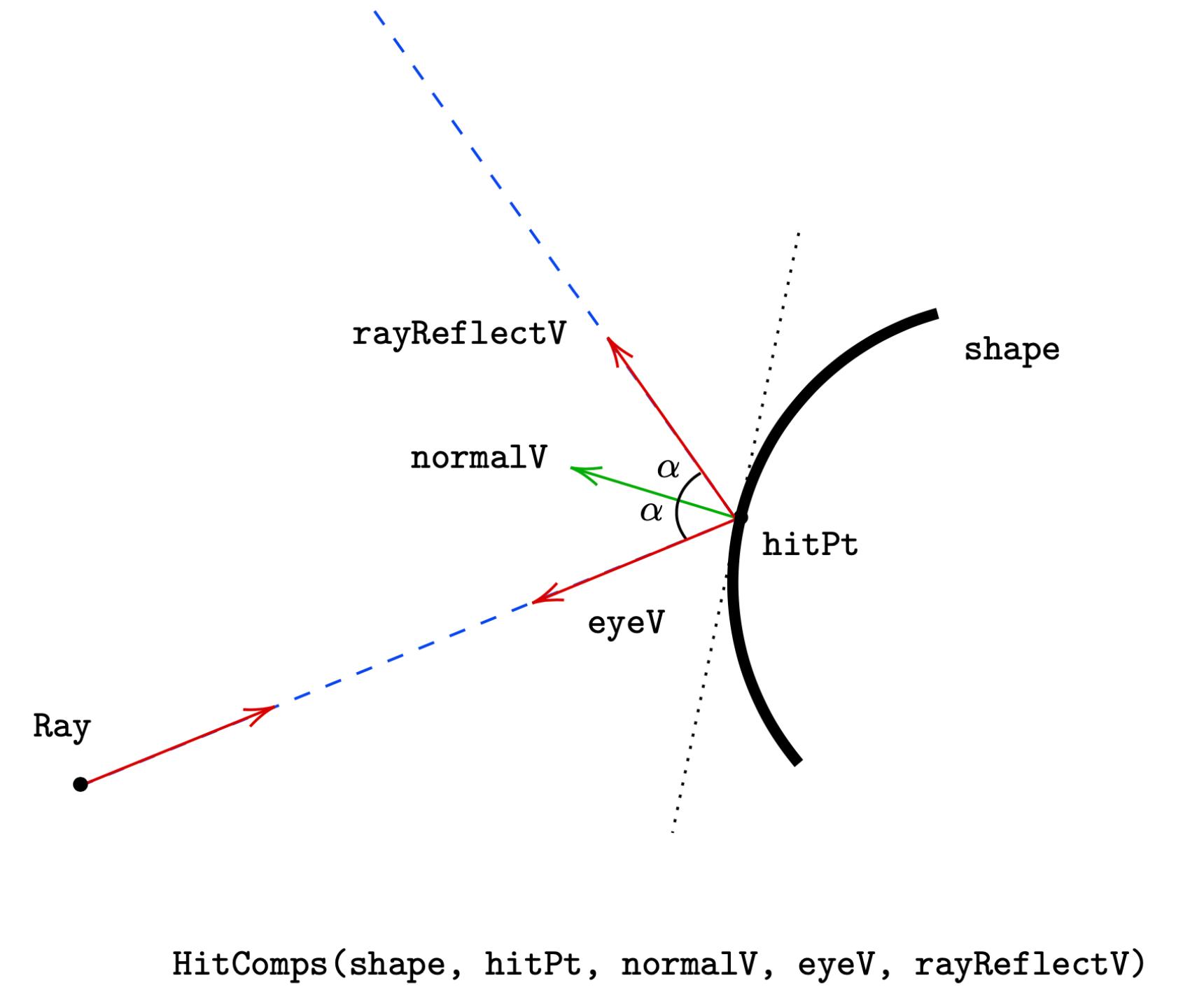
```
trait Live extends WorldModule {  
    val worldTopologyModule: WorldTopologyModule.Service[Any]  
    val worldHitCompsModule: WorldHitCompsModule.Service[Any]  
    val phongReflectionModule: PhongReflectionModule.Service[Any]  
    val worldReflectionModule: WorldReflectionModule.Service[Any]
```

# Reflective surfaces

## Use WorldReflection

```
trait Live extends WorldModule {
  val worldTopologyModule: WorldTopologyModule.Service[Any]
  val worldHitCompsModule: WorldHitCompsModule.Service[Any]
  val phongReflectionModule: PhongReflectionModule.Service[Any]
  val worldReflectionModule: WorldReflectionModule.Service[Any]

  val worldModule: Service[Any] =
    new Service[Any] {
      def colorForRay(world: World, ray: Ray): ZIO[Any, RayTracerError, Color] =
        {
          /* ... */
          for {
            color <- /* standard computation of color */
            reflectedColor <- worldReflectionModule.reflectedColor(world, hc)
          } yield color + reflectedColor
        }
    }
}
```

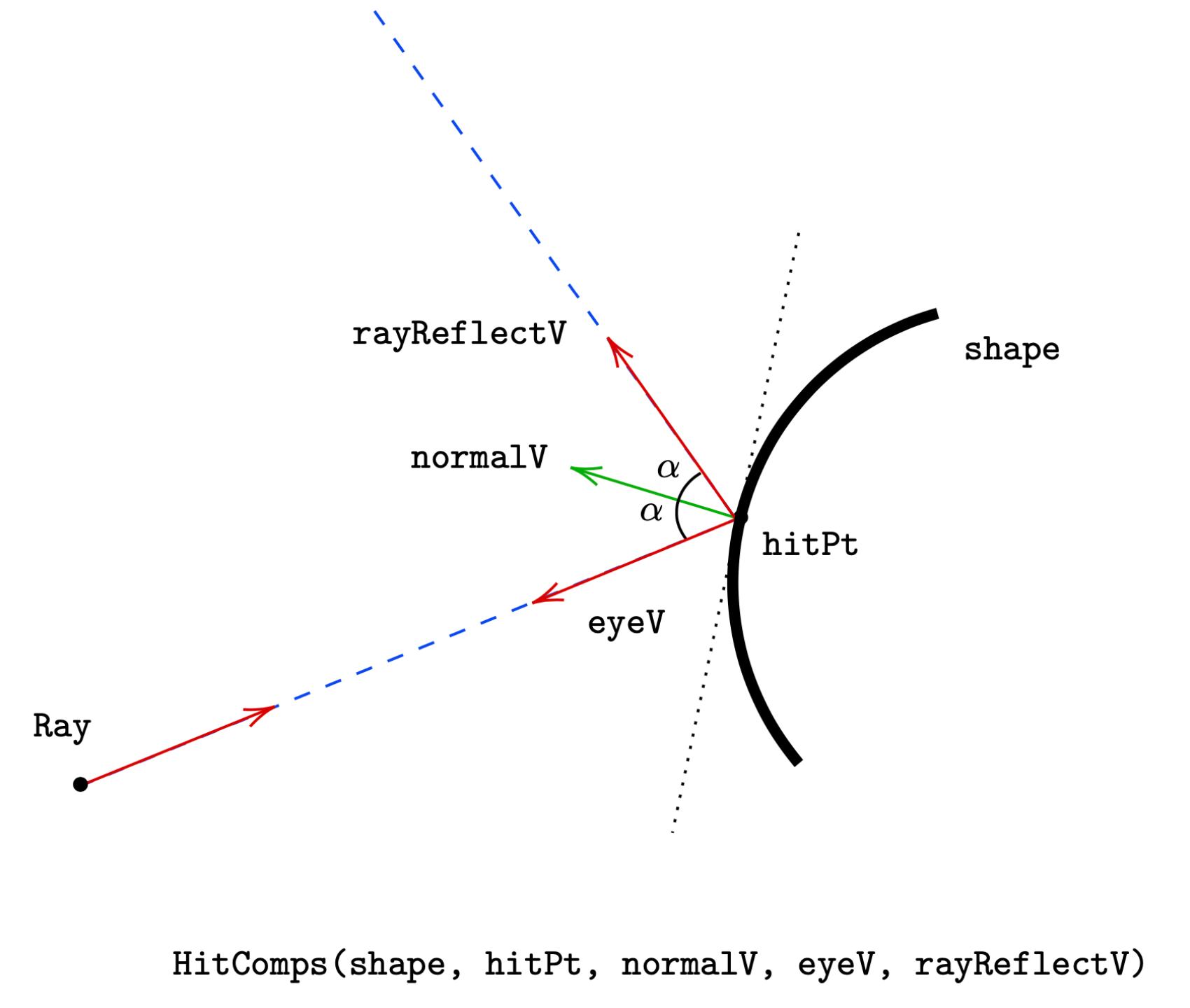


# Reflective surfaces

## Use WorldReflection

```
trait Live extends WorldModule {
  val worldTopologyModule: WorldTopologyModule.Service[Any]
  val worldHitCompsModule: WorldHitCompsModule.Service[Any]
  val phongReflectionModule: PhongReflectionModule.Service[Any]
  val worldReflectionModule: WorldReflectionModule.Service[Any]

  val worldModule: Service[Any] =
    new Service[Any] {
      def colorForRay(world: World, ray: Ray): ZIO[Any, RayTracerError, Color] =
        {
          /* ... */
          for {
            color <- /* standard computation of color */
            reflectedColor <- worldReflectionModule.reflectedColor(world, hc)
          } yield color + reflectedColor
        }
    }
}
```

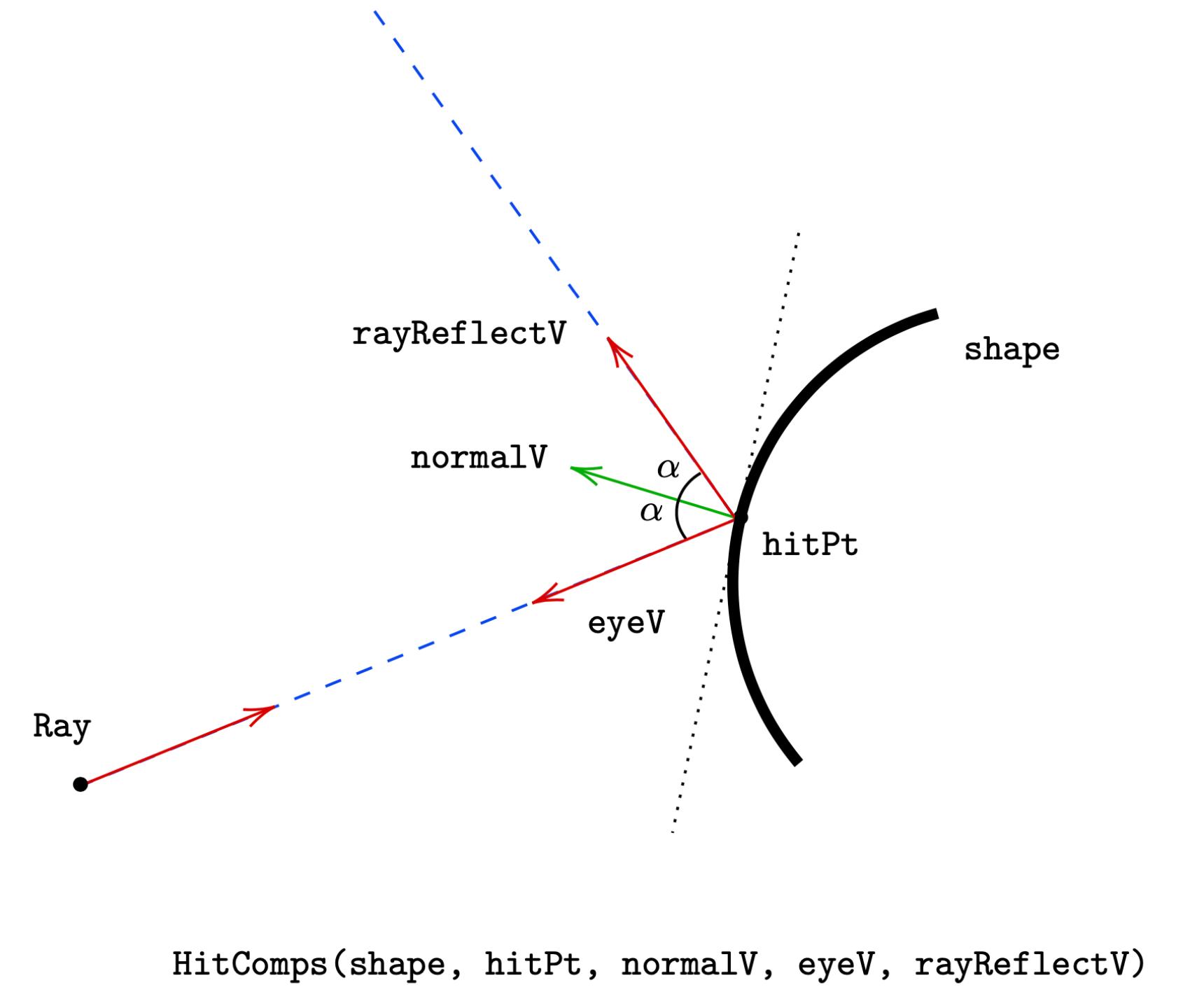


# Reflective surfaces

## Use WorldReflection

```
trait Live extends WorldModule {
  val worldTopologyModule: WorldTopologyModule.Service[Any]
  val worldHitCompsModule: WorldHitCompsModule.Service[Any]
  val phongReflectionModule: PhongReflectionModule.Service[Any]
  val worldReflectionModule: WorldReflectionModule.Service[Any]

  val worldModule: Service[Any] =
    new Service[Any] {
      def colorForRay(world: World, ray: Ray): ZIO[Any, RayTracerError, Color] =
      {
        /* ... */
        for {
          color <- /* standard computation of color */
          reflectedColor <- worldReflectionModule.reflectedColor(world, hc)
        } yield color + reflectedColor
      }
    }
}
```

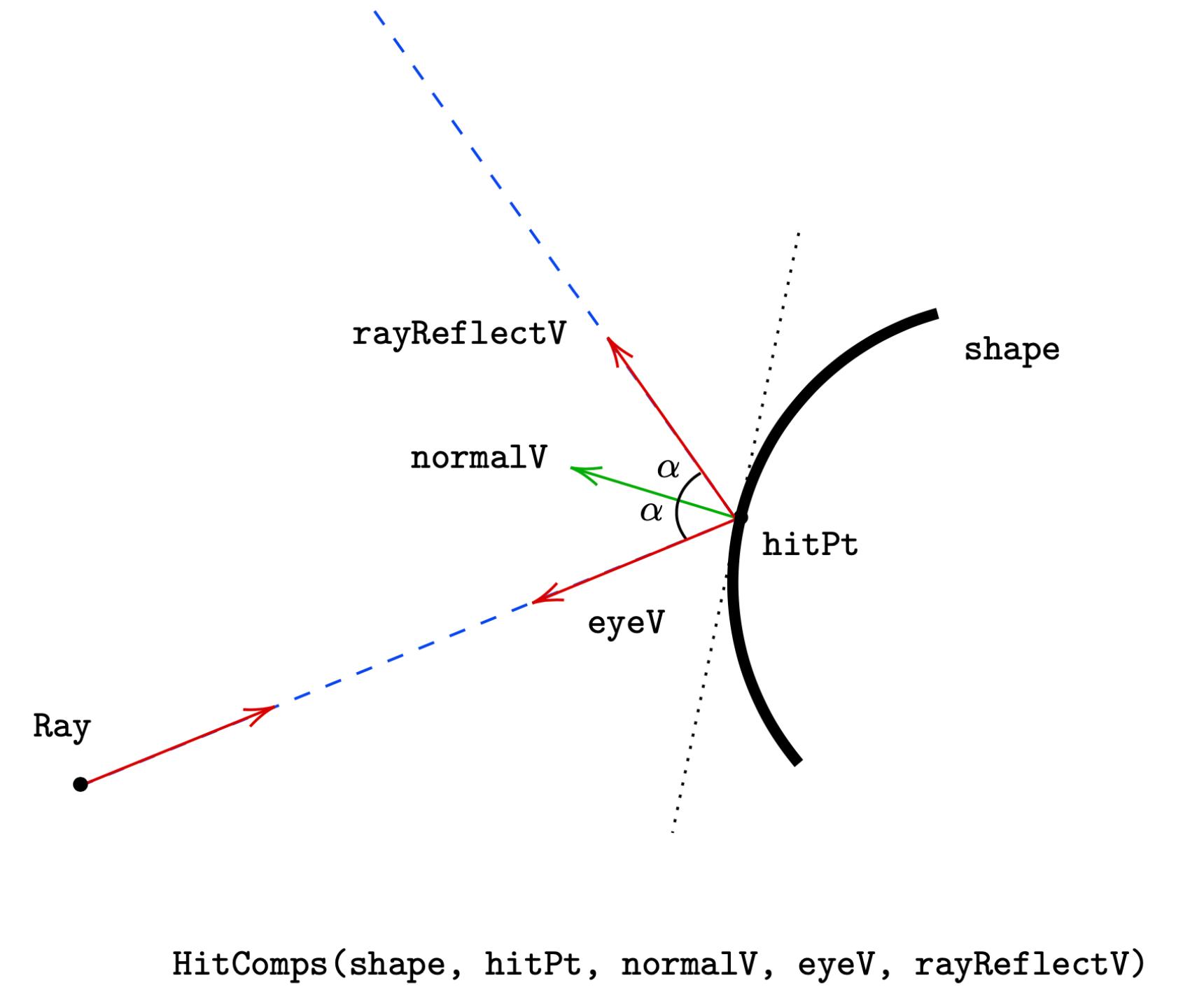


# Reflective surfaces

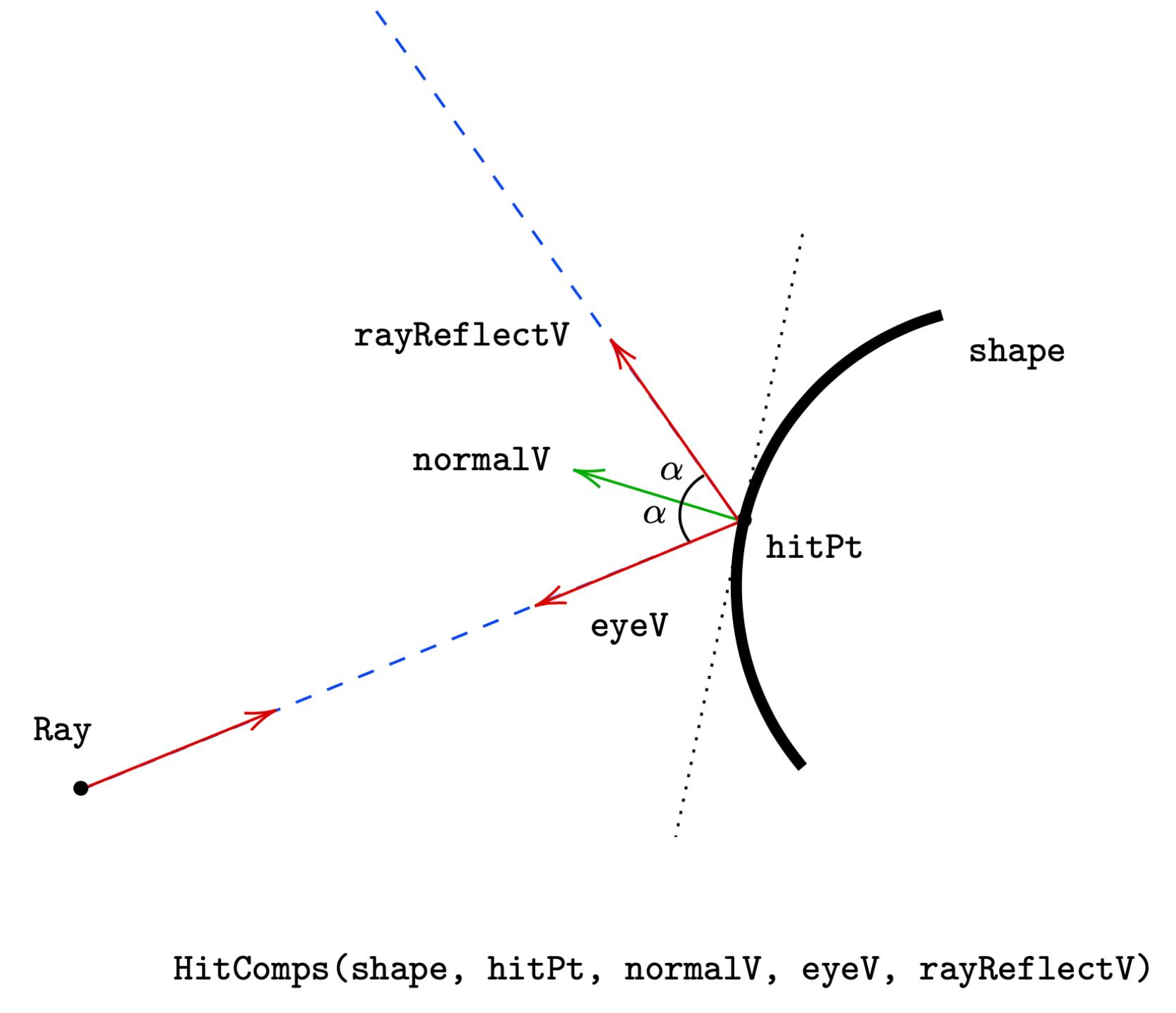
## Use WorldReflection

```
trait Live extends WorldModule {
  val worldTopologyModule: WorldTopologyModule.Service[Any]
  val worldHitCompsModule: WorldHitCompsModule.Service[Any]
  val phongReflectionModule: PhongReflectionModule.Service[Any]
  val worldReflectionModule: WorldReflectionModule.Service[Any]

  val worldModule: Service[Any] =
    new Service[Any] {
      def colorForRay(world: World, ray: Ray): ZIO[Any, RayTracerError, Color] =
      {
        /* ... */
        for {
          color <- /* standard computation of color */
          reflectedColor <- worldReflectionModule.reflectedColor(world, hc)
        } yield color + reflectedColor
      }
    }
}
```



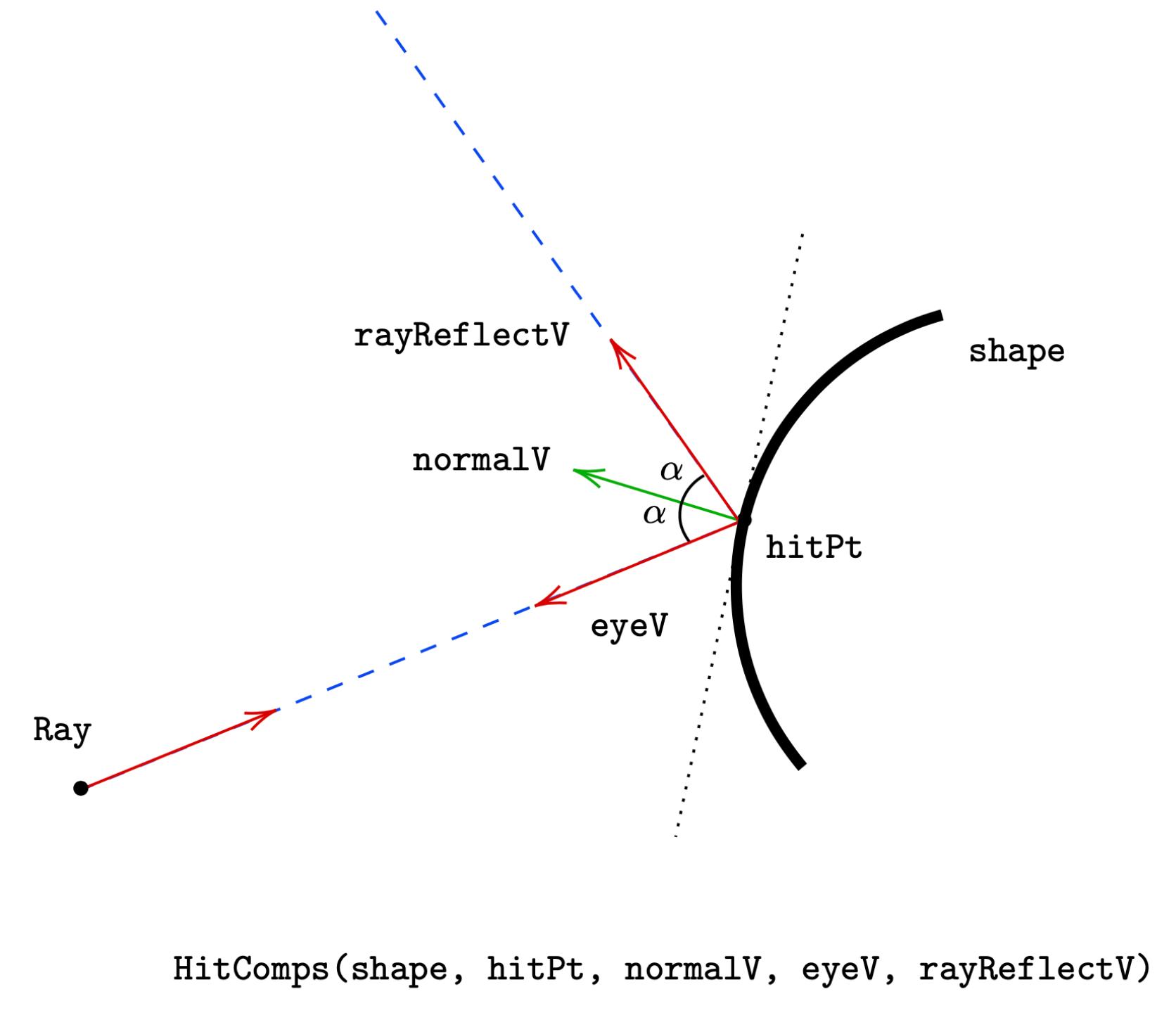
# Implement WorldReflection Live



```
trait Live extends WorldReflectionModule {
  val worldModule: WorldModule.Service[Any]

  val worldReflectionModule = new WorldReflectionModule.Service[Any] {
    def reflectedColor(world: World, hitComps: HitComps, remaining: Int): ZIO[Any, RayTracerError, Color] =
      if (hitComps.shape.material.reflective == 0) {
        UIO(Color.black)
      } else {
        val reflRay = Ray(hitComps.overPoint, hitComps.rayReflectV)
        worldModule.colorForRay(world, reflRay, remaining).map(c =>
          c * hitComps.shape.material.reflective
        )
      }
  }
}
```

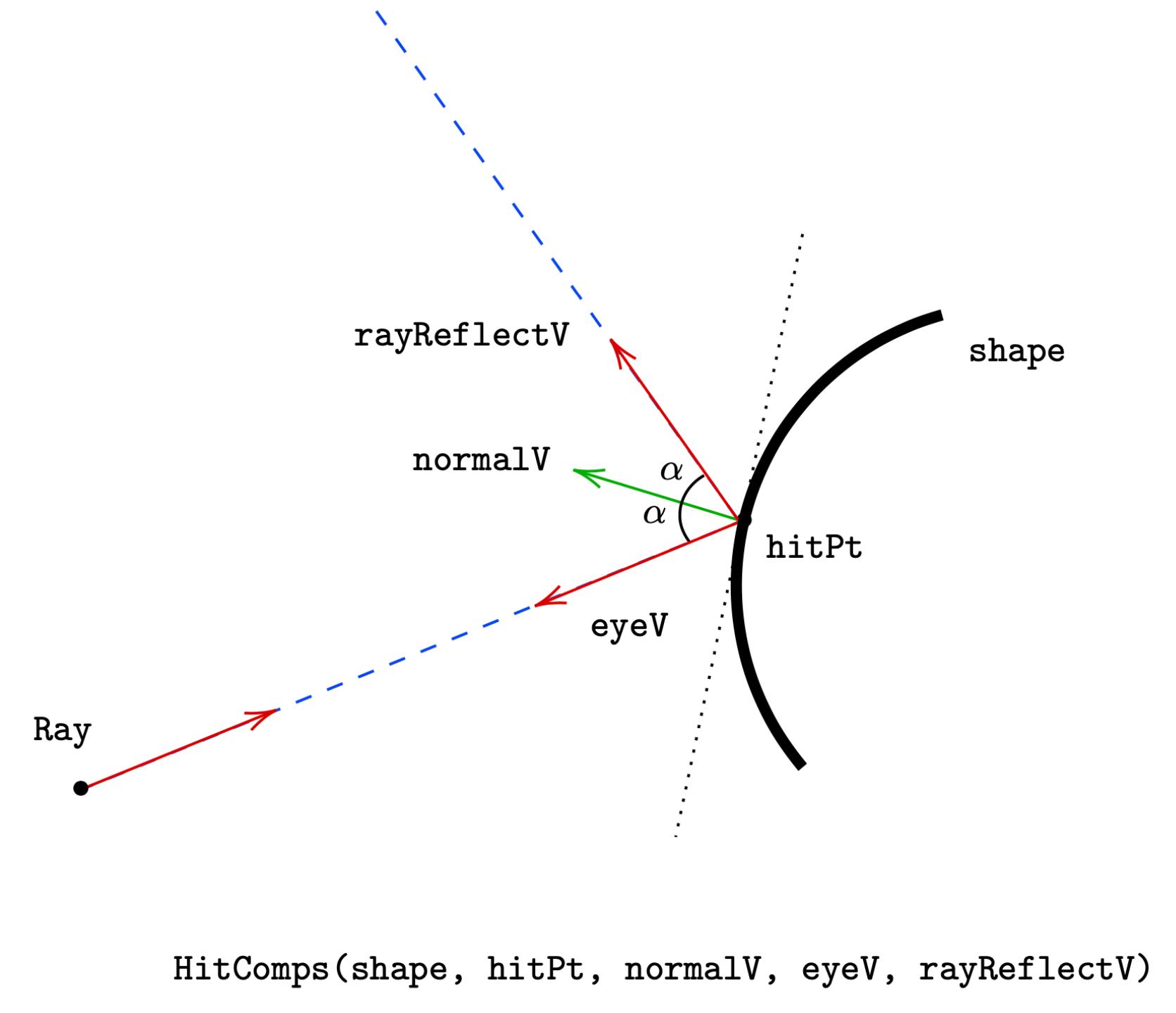
# Implement WorldReflection Live



```
trait Live extends WorldReflectionModule {
  val worldModule: WorldModule.Service[Any]

  val worldReflectionModule = new WorldReflectionModule.Service[Any] {
    def reflectedColor(world: World, hitComps: HitComps, remaining: Int): ZIO[Any, RayTracerError, Color] =
      if (hitComps.shape.material.reflective == 0) {
        UIO(Color.black)
      } else {
        val reflRay = Ray(hitComps.overPoint, hitComps.rayReflectV)
        worldModule.colorForRay(world, reflRay, remaining).map(c =>
          c * hitComps.shape.material.reflective
        )
      }
  }
}
```

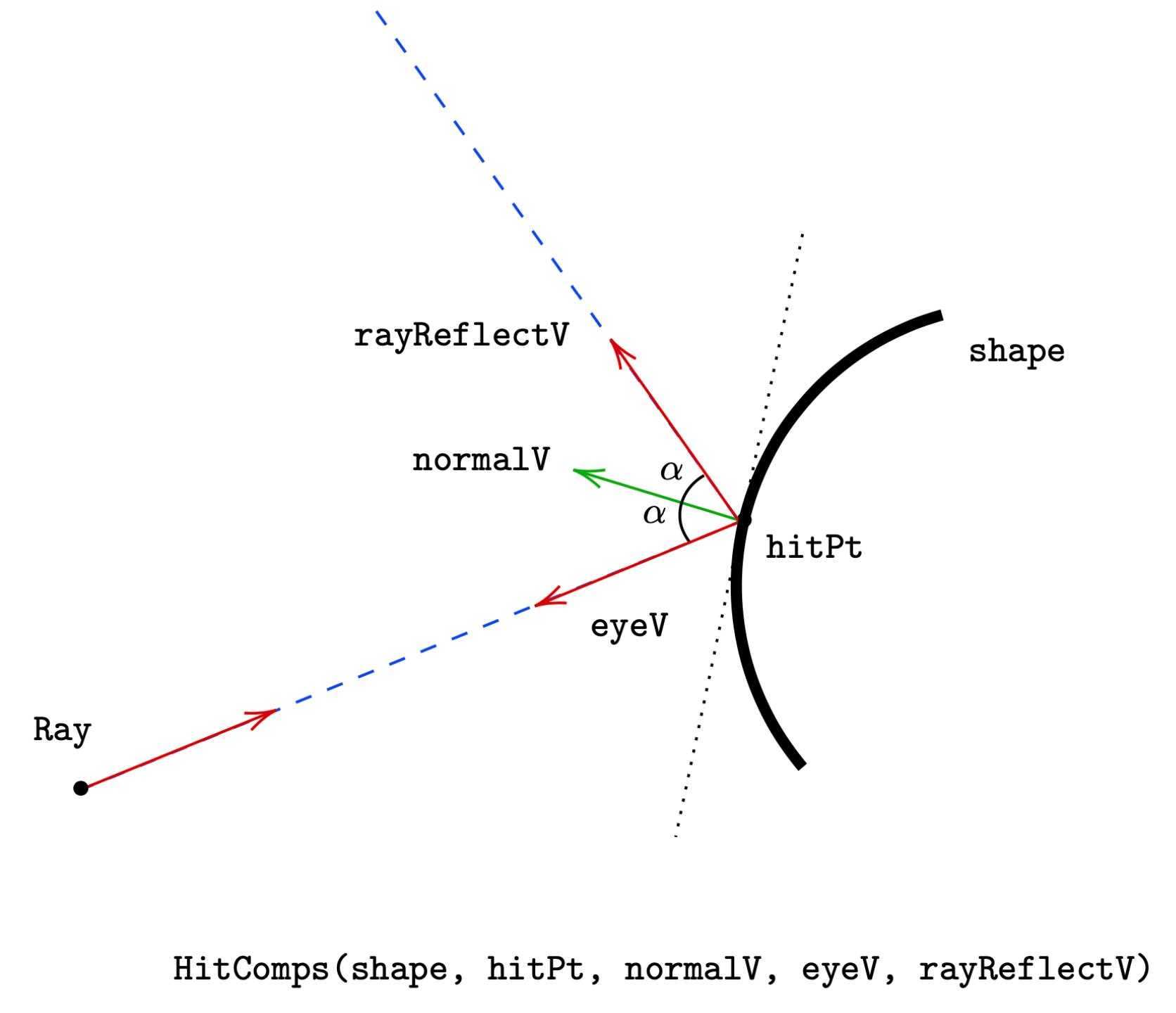
# Implement WorldReflection Live



```
trait Live extends WorldReflectionModule {
    val worldModule: WorldModule.Service[Any]

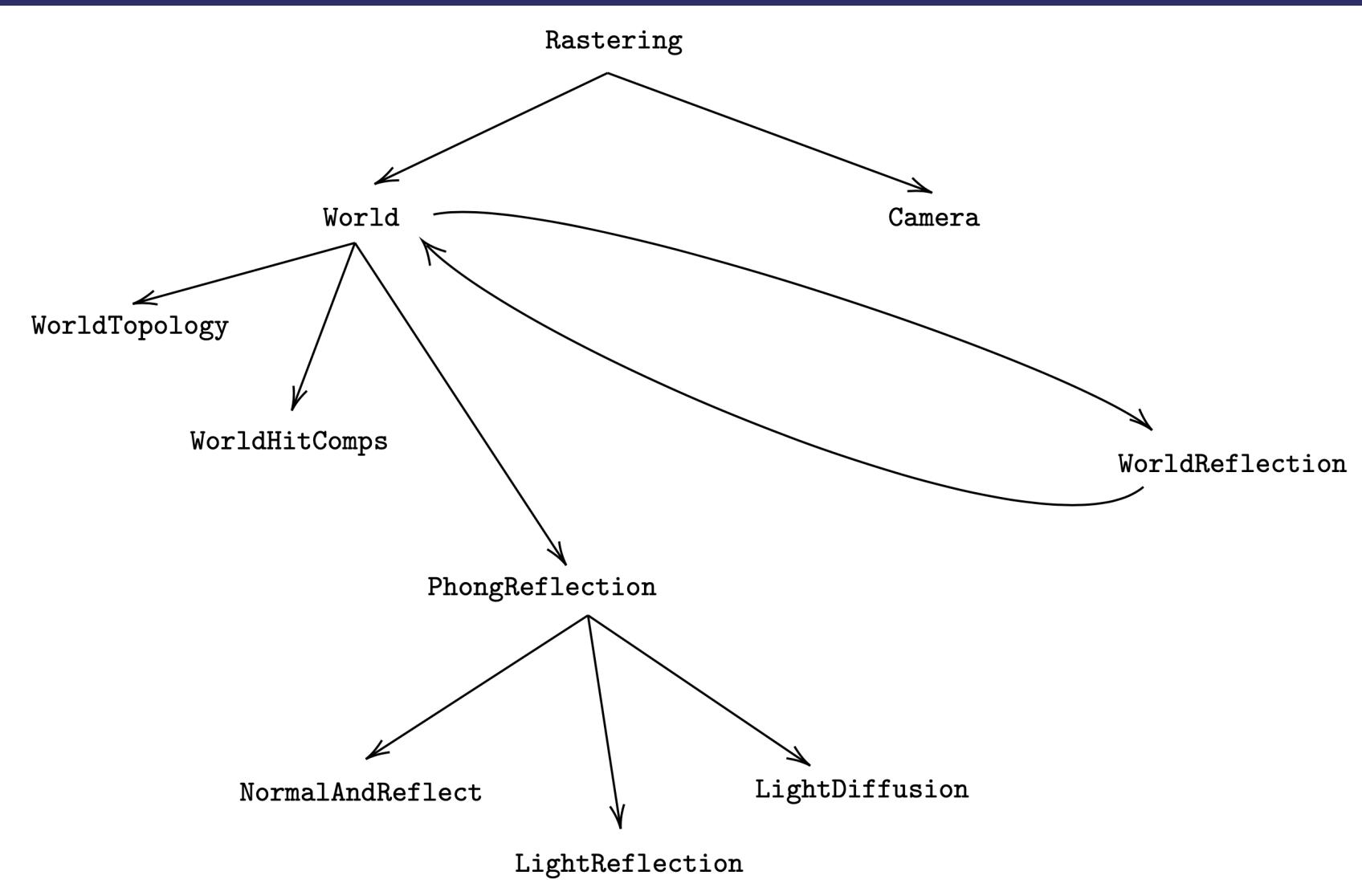
    val worldReflectionModule = new WorldReflectionModule.Service[Any] {
        def reflectedColor(world: World, hitComps: HitComps, remaining: Int): ZIO[Any, RayTracerError, Color] =
            if (hitComps.shape.material.reflective == 0) {
                UIO(Color.black)
            } else {
                val reflRay = Ray(hitComps.overPoint, hitComps.rayReflectV)
                worldModule.colorForRay(world, reflRay, remaining).map(c =>
                    c * hitComps.shape.material.reflective
                )
            }
    }
}
```

# Implement WorldReflection Live



```
trait Live extends WorldReflectionModule {
  val worldModule: WorldModule.Service[Any]

  val worldReflectionModule = new WorldReflectionModule.Service[Any] {
    def reflectedColor(world: World, hitComps: HitComps, remaining: Int): ZIO[Any, RayTracerError, Color] =
      if (hitComps.shape.material.reflective == 0) {
        UIO(Color.black)
      } else {
        val reflRay = Ray(hitComps.overPoint, hitComps.rayReflectV)
        worldModule.colorForRay(world, reflRay, remaining).map(c =>
          c * hitComps.shape.material.reflective
        )
      }
  }
}
```



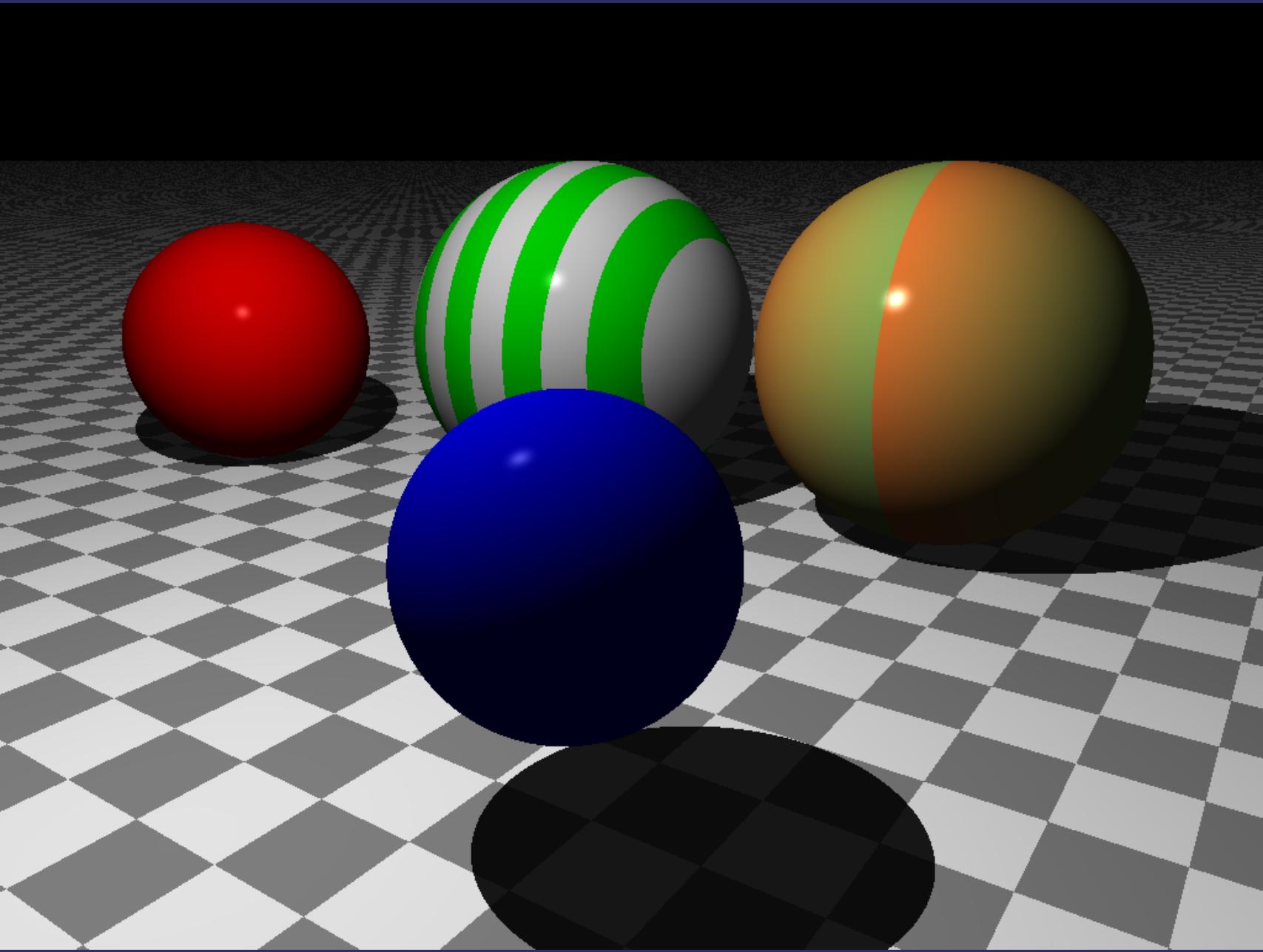
**Reflective surfaces**

**Circular dependency**

# Implement WorldReflection Dummy

```
trait NoReflection extends WorldReflectionModule {
    val worldReflectionModule = new WorldReflectionModule.Service[Any] {
        def reflectedColor(
            world: World,
            hitComps: HitComps,
            remaining: Int
        ): ZIO[Any, RayTracerError, Color] = UIO.succeed(Color.black)
    }
}
```

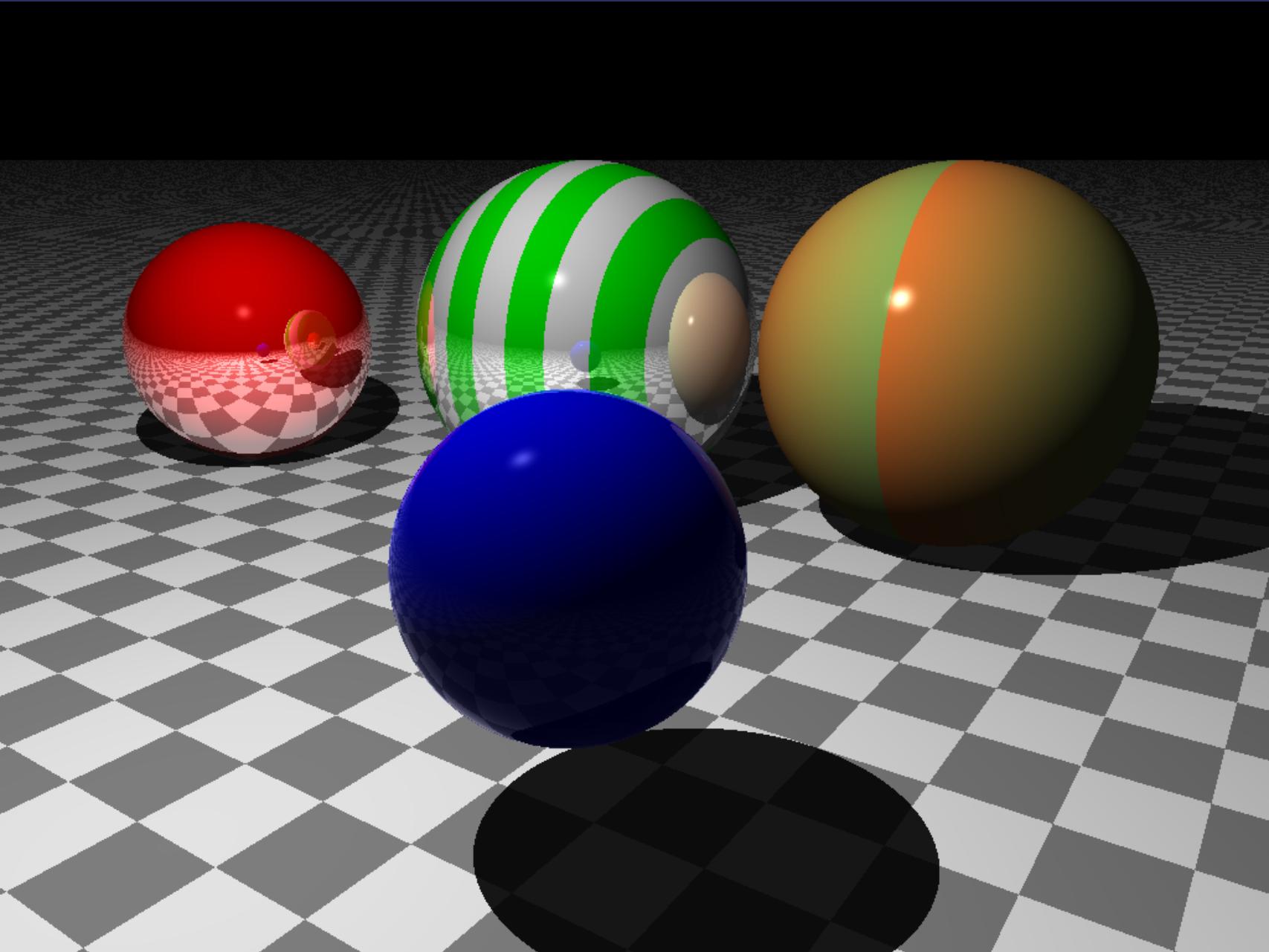
# Spheres



- Red: reflective = 0.9
- Green/white: reflective = 0.6
- Blue: reflective = 0.9,  
transparency: 1

```
program(  
    from = Pt(57, 20, z),  
    to = Pt(20, 0, 20)  
).provide {  
    new BasicModules  
    with PhongReflectionModule.Live  
    with WorldReflectionModule.NoReflection  
}
```

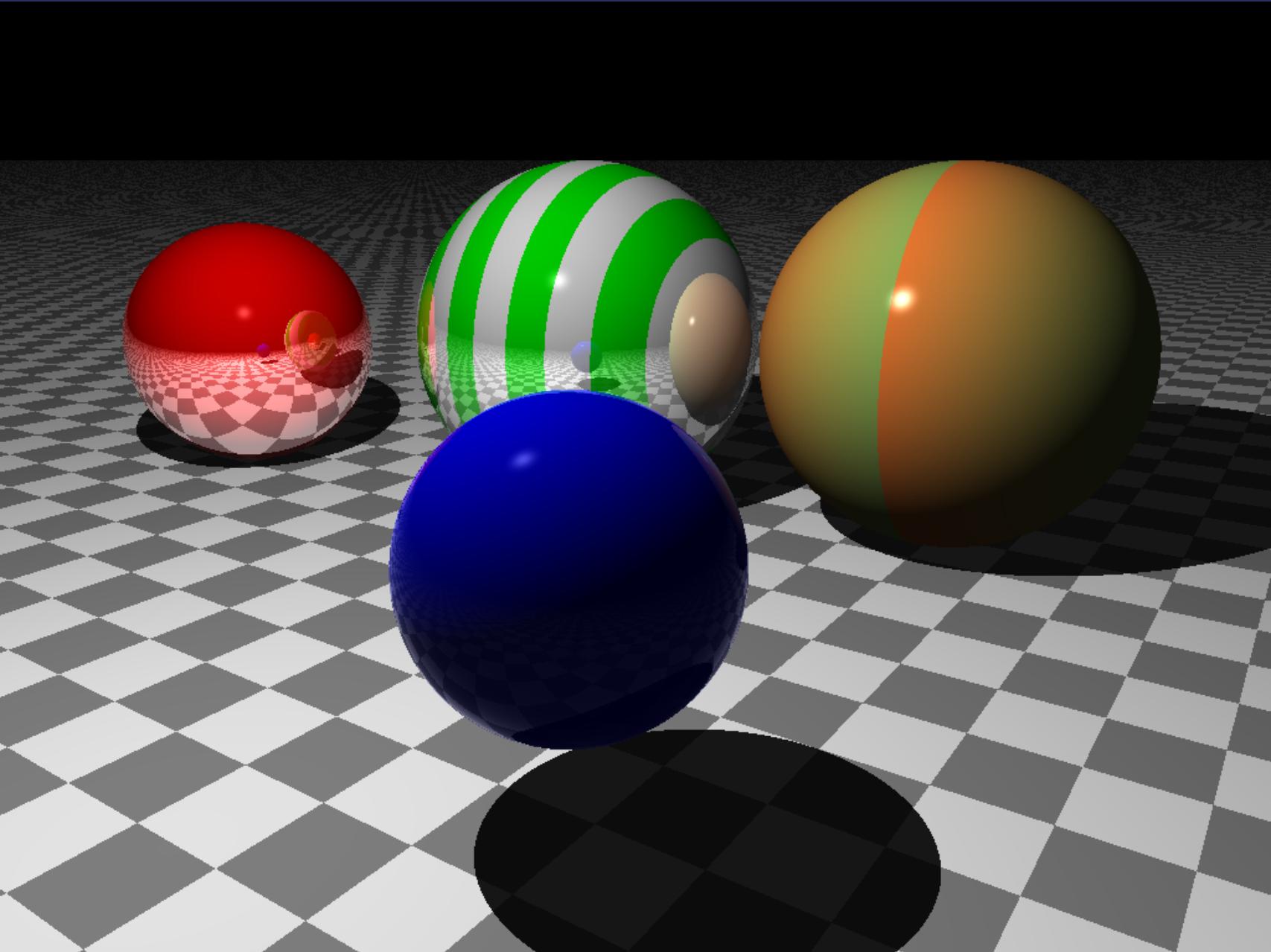
# Spheres



- Red: reflective = 0.9
- Green/white: reflective = 0.6
- Blue: reflective = 0.9,  
transparency: 1

```
program(  
    from = Pt(57, 20, z),  
    to = Pt(20, 0, 20)  
).provide {  
    new BasicModules  
    with PhongReflectionModule.Live  
    with WorldReflectionModule.Live  
}
```

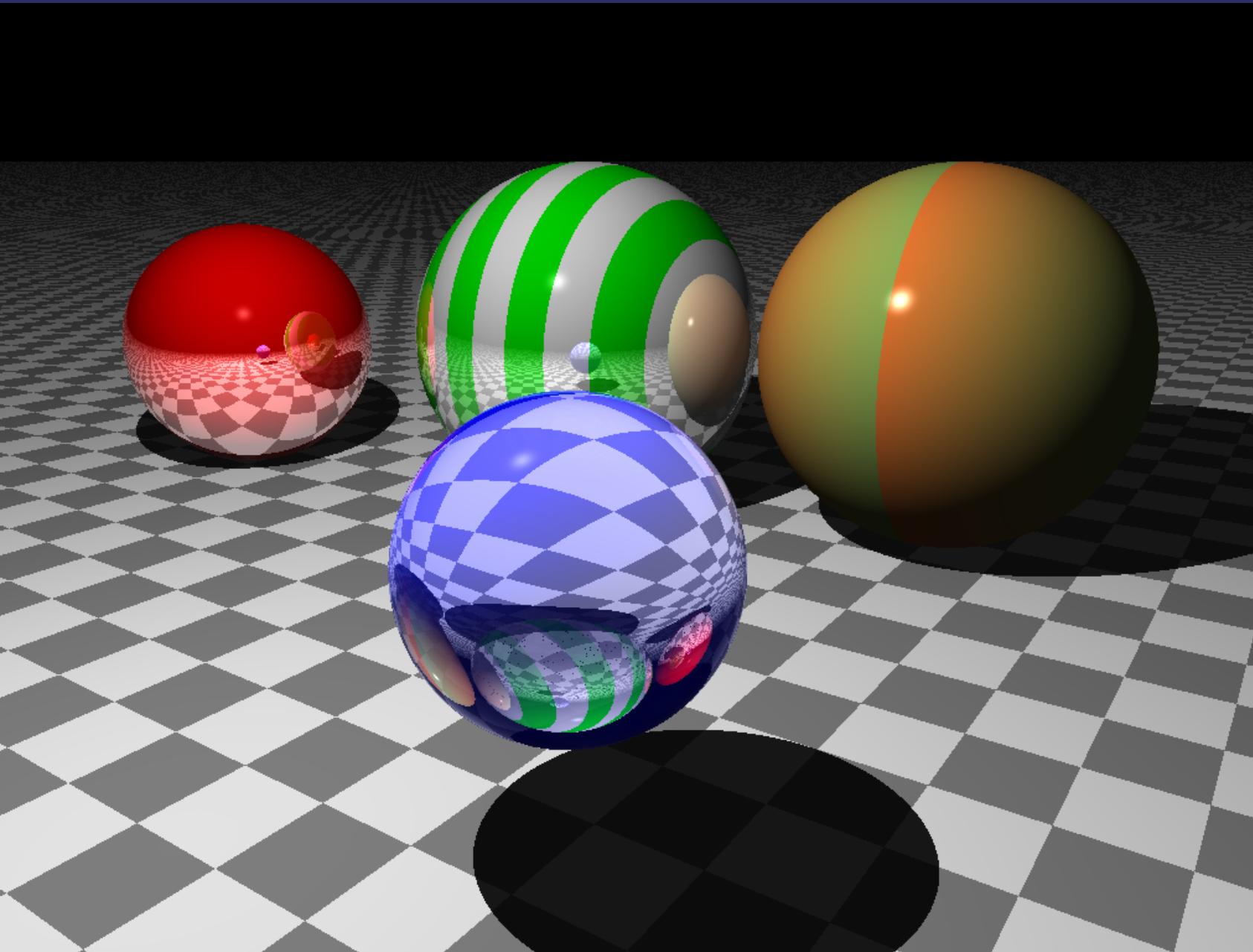
# Spheres



- Red: reflective = 0.9
- Green/white: reflective = 0.6
- Blue: reflective = 0.9,  
transparency: 1

```
program(  
    from = Pt(57, 20, z),  
    to = Pt(20, 0, 20)  
).provide {  
    new BasicModules  
    with PhongReflectionModule.Live  
    with WorldReflectionModule.Live  
    with WorldRefractionModule.NoRefraction  
}
```

# Spheres LiveReflection, LiveRefraction



- Red: reflective = 0.9
- Green/white: reflective = 0.6
- Blue: reflective = 0.9,  
transparency: 1

```
program(  
    from = Pt(57, 20, z),  
    to = Pt(20, 0, 20)  
).provide {  
    new BasicModules  
    with PhongReflectionModule.Live  
    with WorldReflectionModule.Live  
    with WorldRefractionModule.Live  
}
```

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

→ Do not require HKT, typeclasses, etc

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

- Do not require HKT, typeclasses, etc
- Do not abuse typeclasses

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

- Do not require HKT, typeclasses, etc
- Do not abuse typeclasses
- Can group capabilities

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

- Do not require HKT, typeclasses, etc
- Do not abuse typeclasses
- Can group capabilities
- Can provide capabilities one at a time - provideSome

# Conclusion - Environmental Effects

ZIO[R, E, A]

Build purely functional, testable, modular applications

- Do not require HKT, typeclasses, etc
- Do not abuse typeclasses
- Can group capabilities
- Can provide capabilities one at a time - provideSome
- Are not dependent on implicits (survive refactoring)

# Conclusion - Environmental Effects

## Conclusion - Environmental Effects

→ Low entry barrier, very mechanical 🤖

## Conclusion - Environmental Effects

- Low entry barrier, very mechanical 
- Macros help with boilerplate 

## Conclusion - Environmental Effects

- Low entry barrier, very mechanical 
- Macros help with boilerplate 
- Compiler is your friend 

## Conclusion - Environmental Effects

- Low entry barrier, very mechanical 
- Macros help with boilerplate 
- Compiler is your friend 
- Handle circular dependencies 

## Conclusion - Environmental Effects

- Low entry barrier, very mechanical 
- Macros help with boilerplate 
- Compiler is your friend 
- Handle circular dependencies 
- Try it out! 

## Conclusion - Environmental Effects

- Low entry barrier, very mechanical 
- Macros help with boilerplate 
- Compiler is your friend 
- Handle circular dependencies 
- Try it out! 
- Join ZIO Discord channel 

# Thank you!



@pierangelocecc



<https://github.com/pierangeloc/ray-tracer-zio>