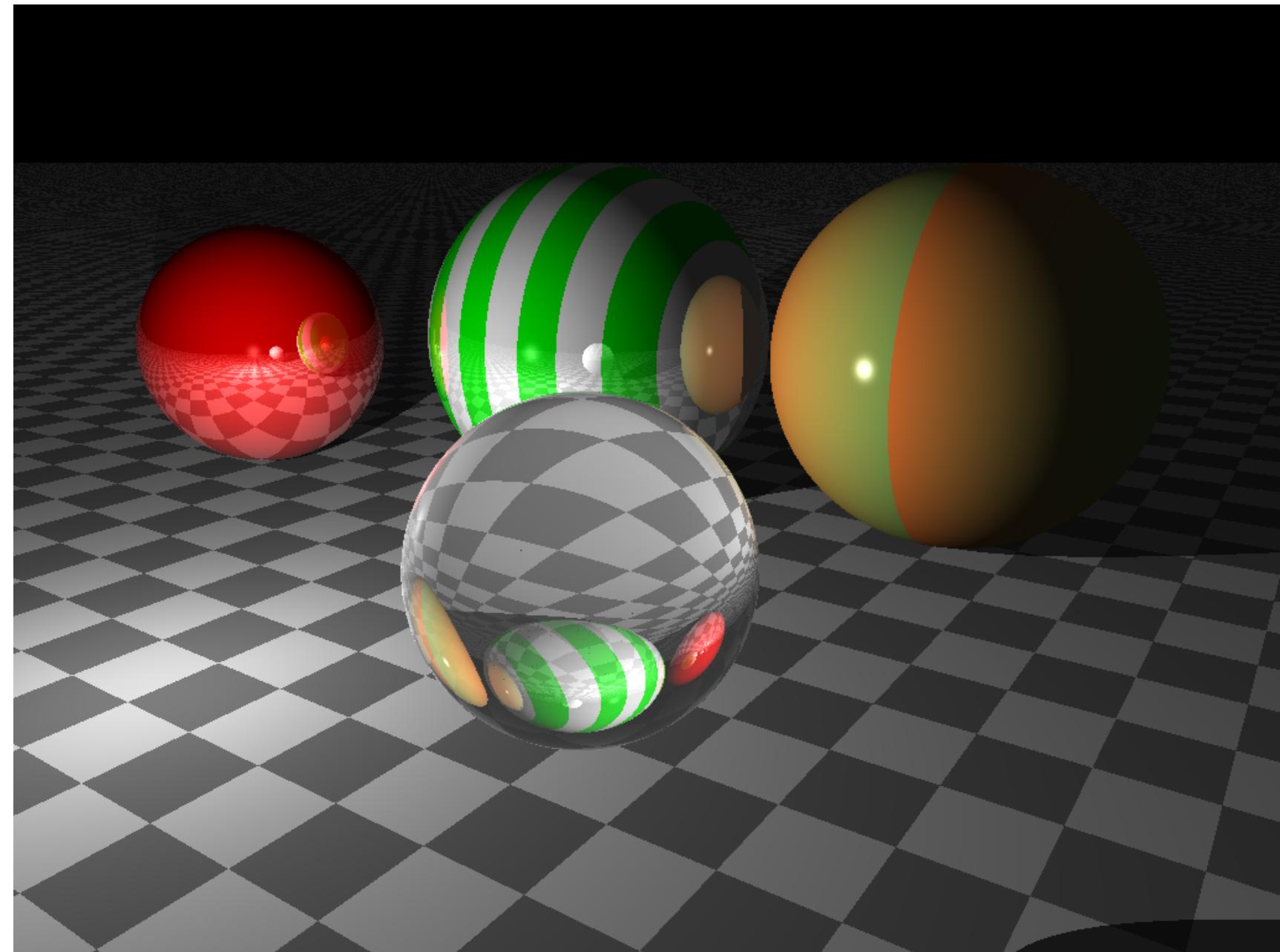


# ZLayer

## A RAY TRACING EXERCISE

PIERANGELO CECCHETTO

Functional Rotterdam  
04 Mar 2020



# ZIO-101

ZIO[-R, +E, +A]



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



R => Either[E, A]

# ZIO-101: PROGRAMS

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values

```
val prg: ZIO[Console with Random, Nothing, Long] = for {
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]
} yield n

val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))

val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))

val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: PROGRAMS

- ▶ ZIO programs are values
- ▶ Concurrency based on fibers (green threads)

```
val prg: ZIO[Console with Random, Nothing, Long] = for {  
    n <- random.nextLong                      // ZIO[Random, Nothing, Long]  
    _ <- console.putStrLn(s"Extracted $n ") // ZIO[Console, Nothing, Unit]  
} yield n
```

```
val allNrs: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAll(List.fill(100)(prg))
```

```
val allNrsPar: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllPar(List.fill(100)(prg))
```

```
val allNrsParN: ZIO[Console with Random, Nothing, List[Long]] = ZIO.collectAllParN(10)(List.fill(100)(prg))
```

# ZIO-101: R MEANS REQUIREMENT

val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

# ZIO-101: R MEANS REQUIREMENT

val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

# ZIO-101: R MEANS REQUIREMENT

val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

# ZIO-101: R MEANS REQUIREMENT

val prg: ZIO[Console with Random, Nothing, Long] = ???

val autonomous: ZIO[Any, Nothing, Long] = ???

val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???

# ZIO-101: REQUIREMENTS ELIMINATION

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

```
val provided: ZIO[Any, Nothing, User] =  
  getUserFromDb.provide(DBConnection(...))
```

```
val user: User = Runtime.default.unsafeRun(provided)
```

# ZIO-101: REQUIREMENTS ELIMINATION

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

```
val provided: ZIO[Any, Nothing, User] =  
  getUserFromDb.provide(DBConnection(...))
```

```
val user: User = Runtime.default.unsafeRun(provided)
```

# ZIO-101: REQUIREMENTS ELIMINATION

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

```
val provided: ZIO[Any, Nothing, User] =  
  getUserFromDb.provide(DBConnection(...))
```

```
val user: User = Runtime.default.unsafeRun(provided)
```

# ZIO-101: REQUIREMENTS ELIMINATION

```
val getUserFromDb: ZIO[DBConnection, Nothing, User] = ???
```

```
val provided: ZIO[Any, Nothing, User] =  
  getUserFromDb.provide(DBConnection(...))
```

```
val user: User = Runtime.default.unsafeRun(provided)
```

# ZIO-101: MODULES

**Example: a module to collect metrics<sup>1</sup>**

```
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
    ZIO.accessM(_.get.inc(label))
}
}
```

<sup>1</sup>ZIO 1.0.0-RC18

# ZIO-101: MODULES

Example: a module to collect metrics<sup>1</sup>

```
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
    ZIO.accessM(_.get.inc(label))
}
}
```

<sup>1</sup>ZIO 1.0.0-RC18

# ZIO-101: MODULES

Example: a module to collect metrics<sup>1</sup>

```
type Metrics = Has[Metrics.Service]
object Metrics {
  trait Service {
    def inc(label: String): IO[Nothing, Unit]
  }

  //accessor method
  def inc(label: String): ZIO[Metrics, Nothing, Unit] =
    ZIO.accessM(_.get.inc(label))
}
}
```

<sup>1</sup>ZIO 1.0.0-RC18

# ZIO-101: MODULES

## Example: a module for logging

```
type Log = Has[Log.Service]
object Log {
    trait Service {
        def info(s: String): IO[Nothing, Unit]
        def error(s: String): IO[Nothing, Unit]
    }
    //accessor methods...
}
```

# ZIO-101: MODULES

Write a program using existing modules

```
val prg: ZIO[Metrics with Log, Nothing, Unit] =  
for {  
    _ <- Log.info("Hello")  
    _ <- Metrics.inc("salutation")  
    _ <- Log.info("Rotterdam")  
    _ <- Metrics.inc("subject")  
} yield ()
```

# ZIO-101: THE Has DATA TYPE

**Has[A] is a dependency on a service of type A**

```
val hasLog: Has[Log.Service]          // type Log      = Has[Log.Service]
val hasMetrics: Has[Metrics.Service] // type Metrics = Has[Metrics.Service]
val mix: Log with Metrics = hasLog ++ hasMetrics

//access each service
mix.get[Log.Service].info("Starting the application")
```

# ZIO-101: THE Has DATA TYPE

**Has[A] is a dependency on a service of type A**

```
val hasLog: Has[Log.Service]           // type Log      = Has[Log.Service]
val hasMetrics: Has[Metrics.Service] // type Metrics = Has[Metrics.Service]
val mix: Log with Metrics = hasLog ++ hasMetrics
```

```
//access each service
mix.get[Log.Service].info("Starting the application")
```

# ZIO-101: THE Has DATA TYPE

```
val mix: Log with Metrics = hasLog ++ hasMetrics
```

```
mix.get[Log.Service].info("Starting the application")
```

# ZIO-101: THE Has DATA TYPE

```
val mix: Log with Metrics = hasLog ++ hasMetrics
```

```
mix.get[Log.Service].info("Starting the application")
```

- ▶ Is more powerful than trait mixins

# ZIO-101: THE Has DATA TYPE

```
val mix: Log with Metrics = hasLog ++ hasMetrics
```

```
mix.get[Log.Service].info("Starting the application")
```

- ▶ Is more powerful than trait mixins
- ▶ Is backed by a heterogeneous map ServiceType -> Service

# ZIO-101: THE Has DATA TYPE

```
val mix: Log with Metrics = hasLog ++ hasMetrics
```

```
mix.get[Log.Service].info("Starting the application")
```

- ▶ Is more powerful than trait mixins
- ▶ Is backed by a heterogeneous map ServiceType -> Service
- ▶ Can replace/update services

# ZI0-101: ZLayer

ZLayer[-RIn, +E, +ROut <: Has[\_]]

# ZIO-101: ZLayer

ZLayer[-RIn, +E, +ROut <: Has[\_]]

- ▶ A recipe to build an ROut

# ZIO-101: ZLayer

ZLayer[-RIn, +E, +ROut <: Has[\_]]

- ▶ A recipe to build an ROut
- ▶ Backed by ZManaged: safe acquire/release

# ZIO-101: ZLayer

ZLayer[-RIn, +E, +ROut <: Has[\_]]

- ▶ A recipe to build an ROut
- ▶ Backed by ZManaged: safe acquire/release
- ▶ type NoDeps[+E, +B <: Has[\_]] =  
ZLayer[Any, E, B]

# ZIO-101: ZLayer

## Construct from value

```
val layer: ZLayer.NoDeps[Nothing, UserRepo] =  
  ZLayer.succeed(new UserRepo.Service)
```

# ZIO-101: ZLayer

## Construct from function

```
val layer: ZLayer[Connection, Nothing, UserRepo] =  
  ZLayer.fromFunction { c: Connection =>  
    new UserRepo.Service  
  }
```

# ZIO-101: ZLayer

## Construct from effect

```
import java.sql.Connection
```

```
val e: ZIO[Connection, Error, UserRepo.Service]
```

```
val layer: ZLayer[Connection, Error, UserRepo] =  
  ZLayer.fromEffect(e)
```

# ZIO-101: ZLayer

## Construct from resources

```
import java.sql.Connection

val connectionLayer: ZLayer.NoDeps[Nothing, Has[Connection]] =
  ZLayer.fromAcquireRelease(makeConnection) { c =>
    UIO(c.close())
}
```

# ZIO-101: ZLayer

## Construct from other services

```
val usersLayer: ZLayer[UserRepo with UserValidation, Nothing, BusinessLogic] =  
  
  ZLayer.fromServices[UserRepo.Service, UserValidation.Service] {  
    (repoSvc, validSvc) =>  
      new BusinessLogic.Service {  
        // use repoSvc and validSvc  
      }  
  }
```

# ZIO-101: ZLayer

**Compose horizontally  
(all inputs for all outputs)**

```
val l1: ZLayer[Connection, Nothing, UserRepo]
```

```
val l2: ZLayer[Config, Nothing, AuthPolicy]
```

```
val hor: ZLayer[Connection with Config, Nothing, UserRepo with AuthPolicy] =  
  l1 ++ l2
```

# ZIO-101: ZLayer

**Compose vertically**  
**(output of first for input of second)**

```
val l1: ZLayer[Config, Nothing, Connection]
val l2: ZLayer[Connection, Nothing, UserRepo]
```

```
val ver: ZLayer[Config, Nothing, UserRepo] =
  l1 >>> l2
```

# ZIO-101: ZLayer

## Create module instances

```
type UserRepo = Has[UserRepo.Service]

object UserRepo {
  trait Service {
    def getUser(userId: UserId): IO[DBError, Option[User]]
    def createUser(user: User): IO[DBError, Unit]
  }

  val inMemory: ZLayer.NoDeps[Nothing, UserRepo] = ZLayer.succeed(
    new Service {
      /* impl */
    }
  )

  val db: ZLayer[Connection, Nothing, UserRepo] = ZLayer.fromService { conn: Connection =>
    new Service {
      /* impl uses conn */
    }
  }
}
```

# ZIO-101: ZLayer

## Create module instances

```
type UserRepo = Has[UserRepo.Service]

object UserRepo {
  trait Service {
    def getUser(userId: UserId): IO[DBError, Option[User]]
    def createUser(user: User): IO[DBError, Unit]
  }
}

val inMemory: ZLayer.NoDeps[Nothing, UserRepo] = ZLayer.succeed(
  new Service {
    /* impl */
  }
)

val db: ZLayer[Connection, Nothing, UserRepo] = ZLayer.fromService { conn: Connection =>
  new Service {
    /* impl uses conn */
  }
}
```

# ZIO-101: ZLayer

## Create module instances

```
type UserRepo = Has[UserRepo.Service]

object UserRepo {
  trait Service {
    def getUser(userId: UserId): IO[DBError, Option[User]]
    def createUser(user: User): IO[DBError, Unit]
  }
}

val inMemory: ZLayer.NoDeps[Nothing, UserRepo] = ZLayer.succeed(
  new Service {
    /* impl */
  }
)

val db: ZLayer[Connection, Nothing, UserRepo] = ZLayer.fromService { conn: Connection =>
  new Service {
    /* impl uses conn */
  }
}
```

# ZIO-101: ZLayer

## Provide layers to programs

```
import zio.console.Console
val checkUser: ZIO[UserRepo with AuthPolicy with Console, Nothing, Boolean]

val liveLayer = UserRepo.inMemory ++ AuthPolicy.basicPolicy ++ Console.live

val full: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer
)

val partial: ZIO[Console, Nothing, Boolean] = checkUser.provideSomeLayer(
    UserRepo.inMemory ++ AuthPolicy.basicPolicy
)

val updated: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer ++ UserRepo.postgres
)
```

# ZIO-101: ZLayer

## Provide layers to programs

```
import zio.console.Console
val checkUser: ZIO[UserRepo with AuthPolicy with Console, Nothing, Boolean]

val liveLayer = UserRepo.inMemory ++ AuthPolicy.basicPolicy ++ Console.live

val full: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer
)

val partial: ZIO[Console, Nothing, Boolean] = checkUser.provideSomeLayer(
    UserRepo.inMemory ++ AuthPolicy.basicPolicy
)

val updated: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer ++ UserRepo.postgres
)
```

# ZIO-101: ZLayer

## Provide layers to programs

```
import zio.console.Console
val checkUser: ZIO[UserRepo with AuthPolicy with Console, Nothing, Boolean]

val liveLayer = UserRepo.inMemory ++ AuthPolicy.basicPolicy ++ Console.live

val full: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer
)

val partial: ZIO[Console, Nothing, Boolean] = checkUser.provideSomeLayer(
    UserRepo.inMemory ++ AuthPolicy.basicPolicy
)

val updated: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer ++ UserRepo.postgres
)
```

# ZIO-101: ZLayer

## Provide layers to programs

```
import zio.console.Console
val checkUser: ZIO[UserRepo with AuthPolicy with Console, Nothing, Boolean]

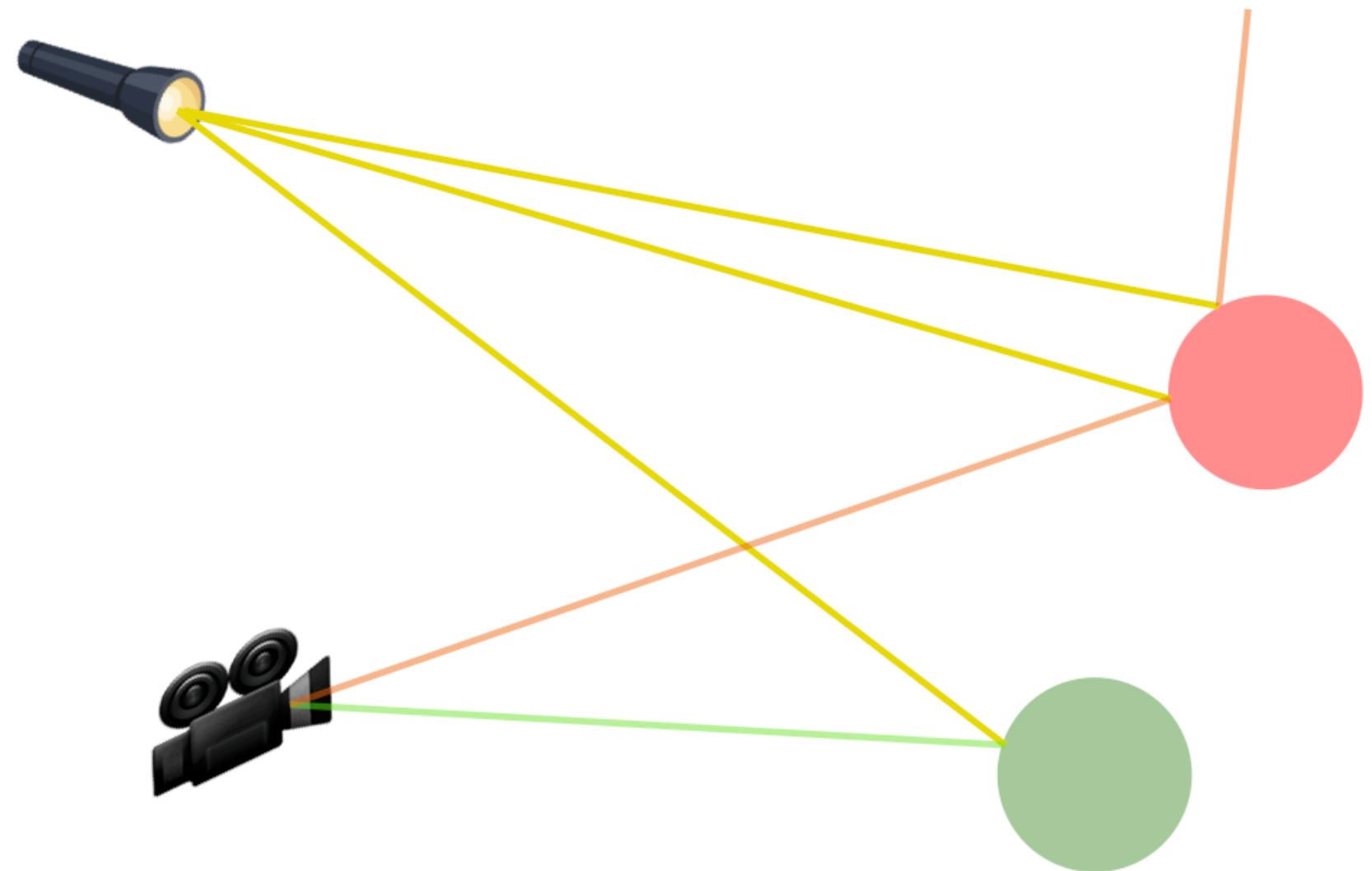
val liveLayer = UserRepo.inMemory ++ AuthPolicy.basicPolicy ++ Console.live

val full: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer
)

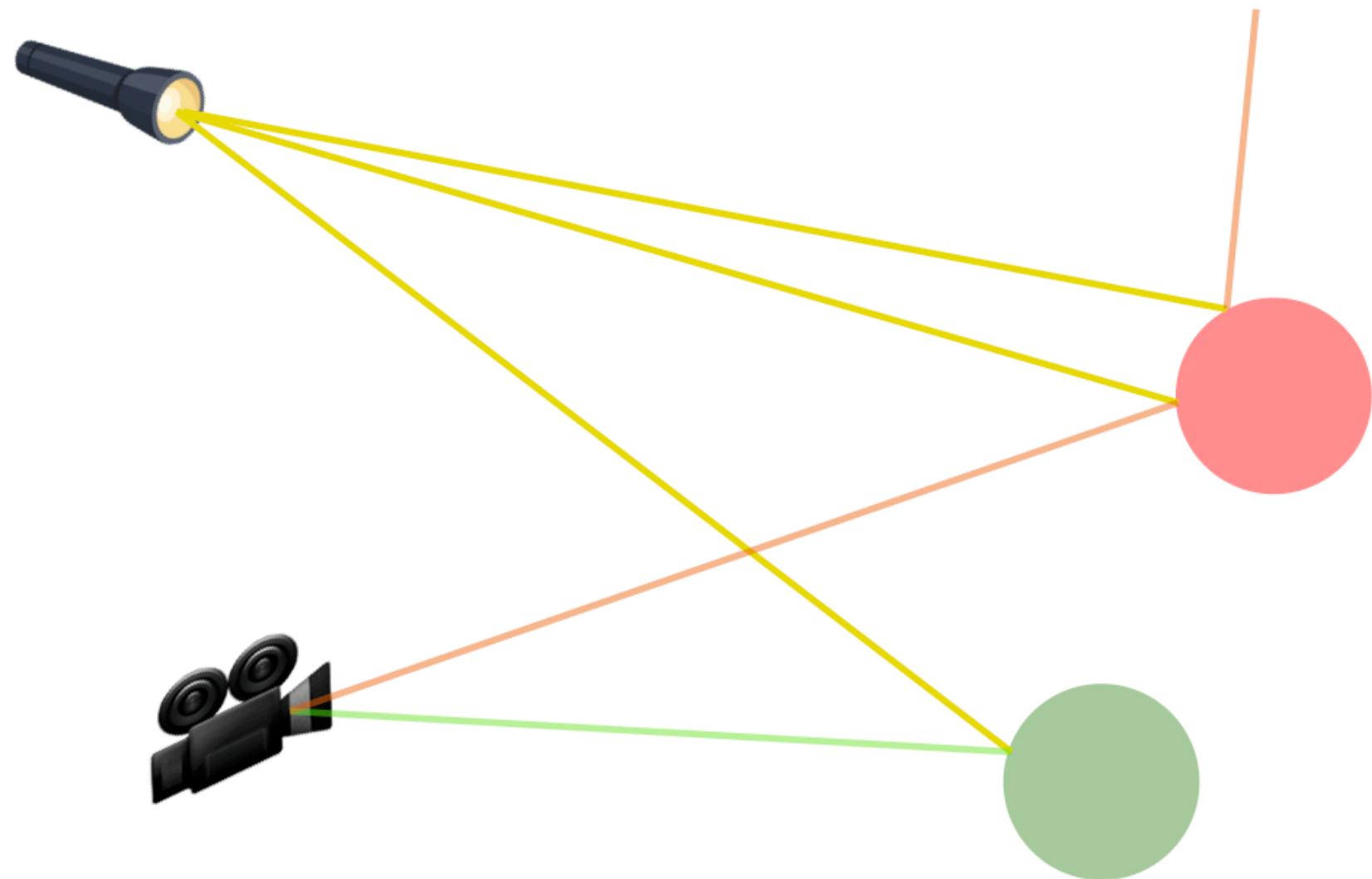
val partial: ZIO[Console, Nothing, Boolean] = checkUser.provideSomeLayer(
    UserRepo.inMemory ++ AuthPolicy.basicPolicy
)

val updated: ZIO[Any, Nothing, Boolean] = checkUser.provideLayer(
    liveLayer ++ UserRepo.postgres
)
```

# RAY TRACING

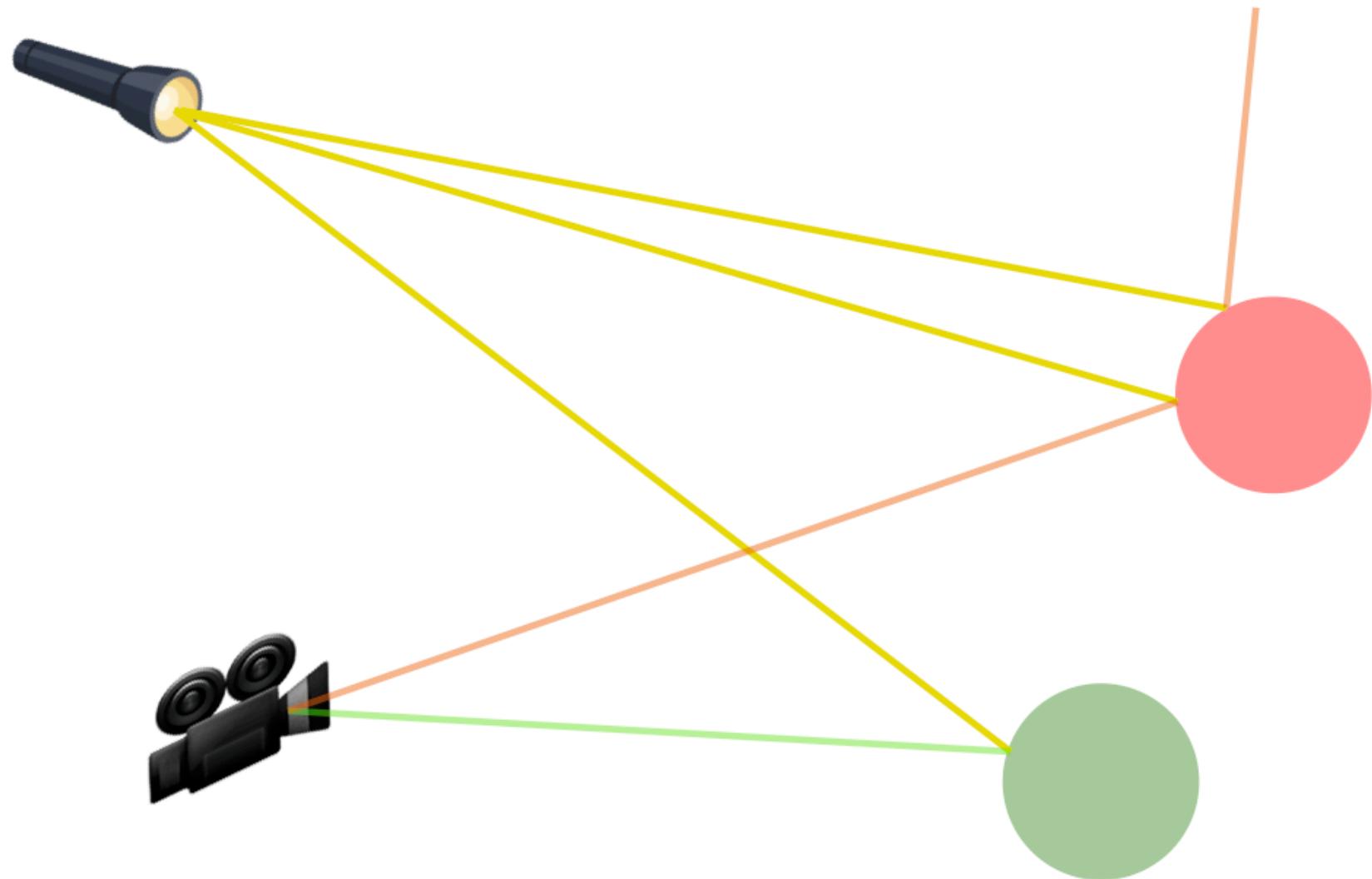


# RAY TRACING



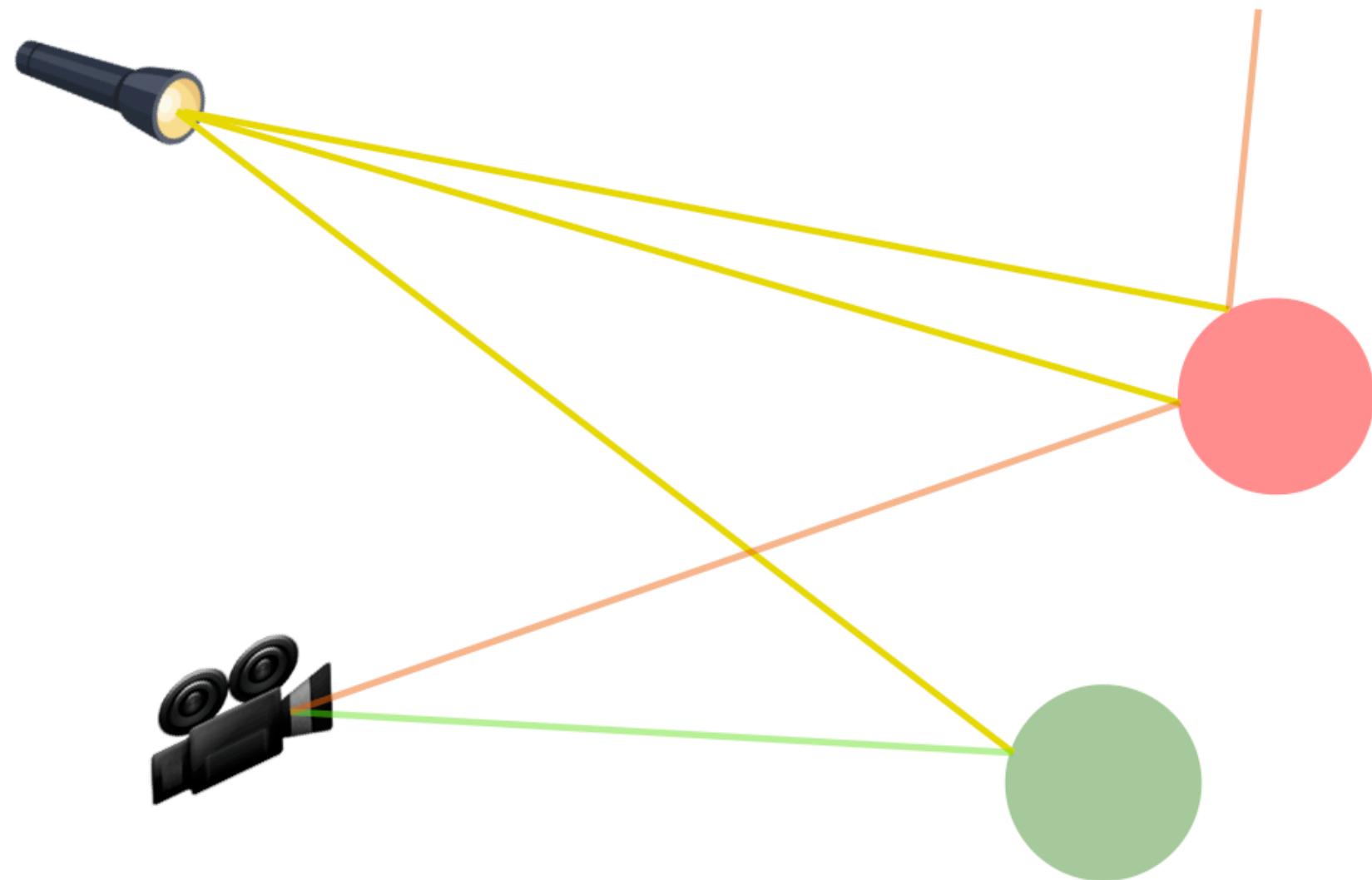
► World (spheres), light source, camera

# RAY TRACING



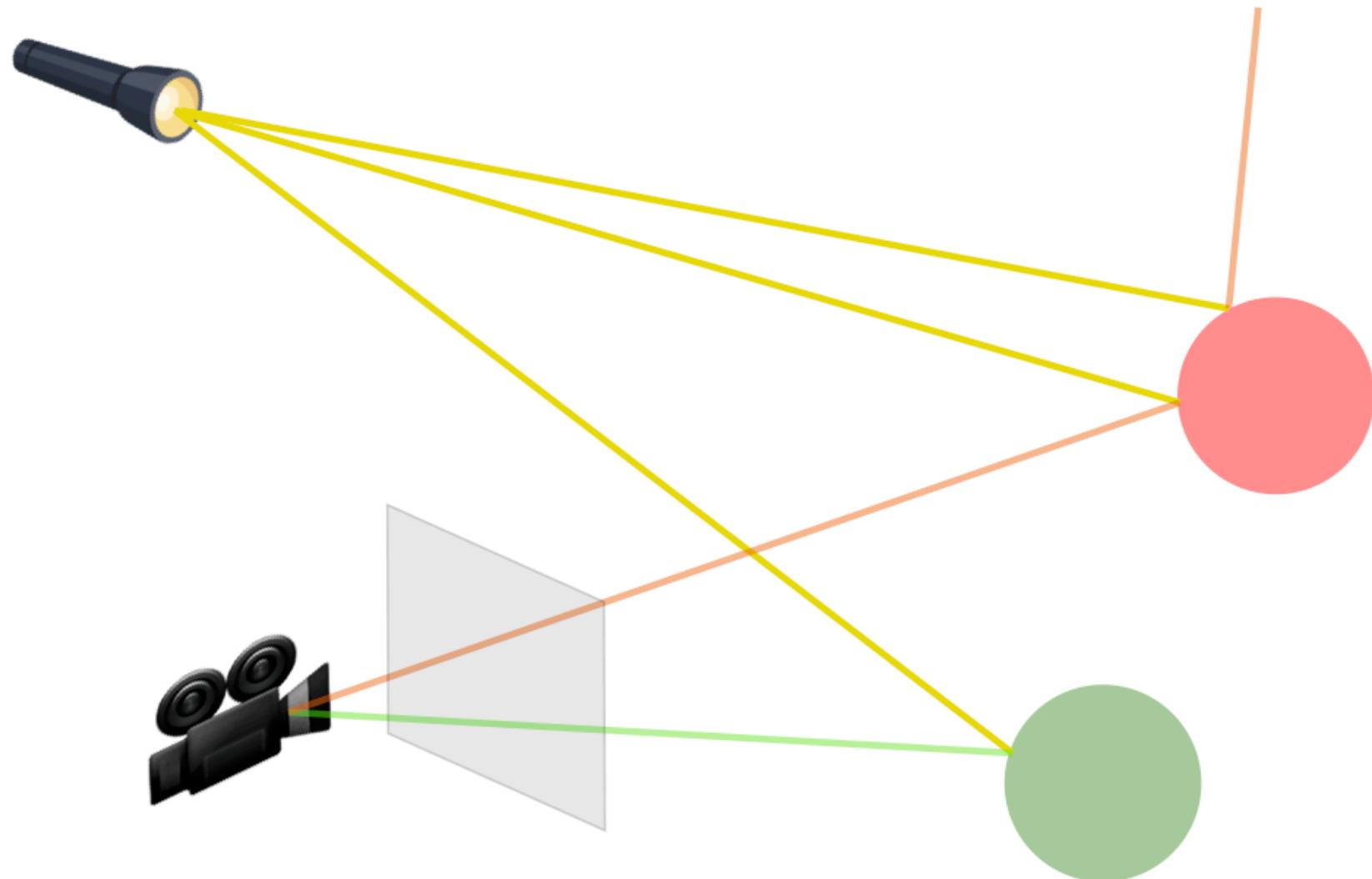
- ▶ World (spheres), light source, camera
- ▶ Incident rays

# RAY TRACING



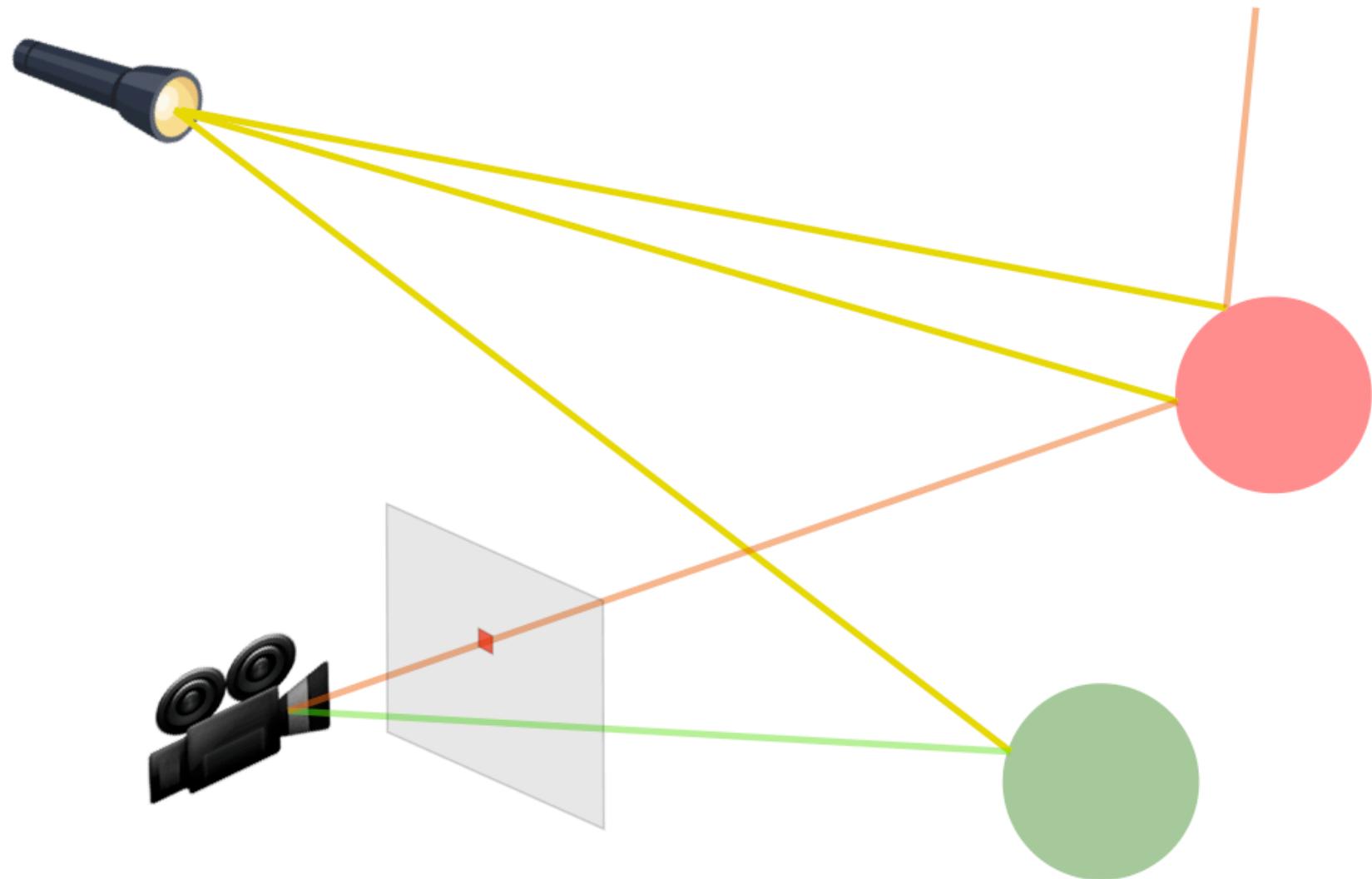
- ▶ World (spheres), light source, camera
- ▶ Incident rays
- ▶ Reflected rays

# RAY TRACING



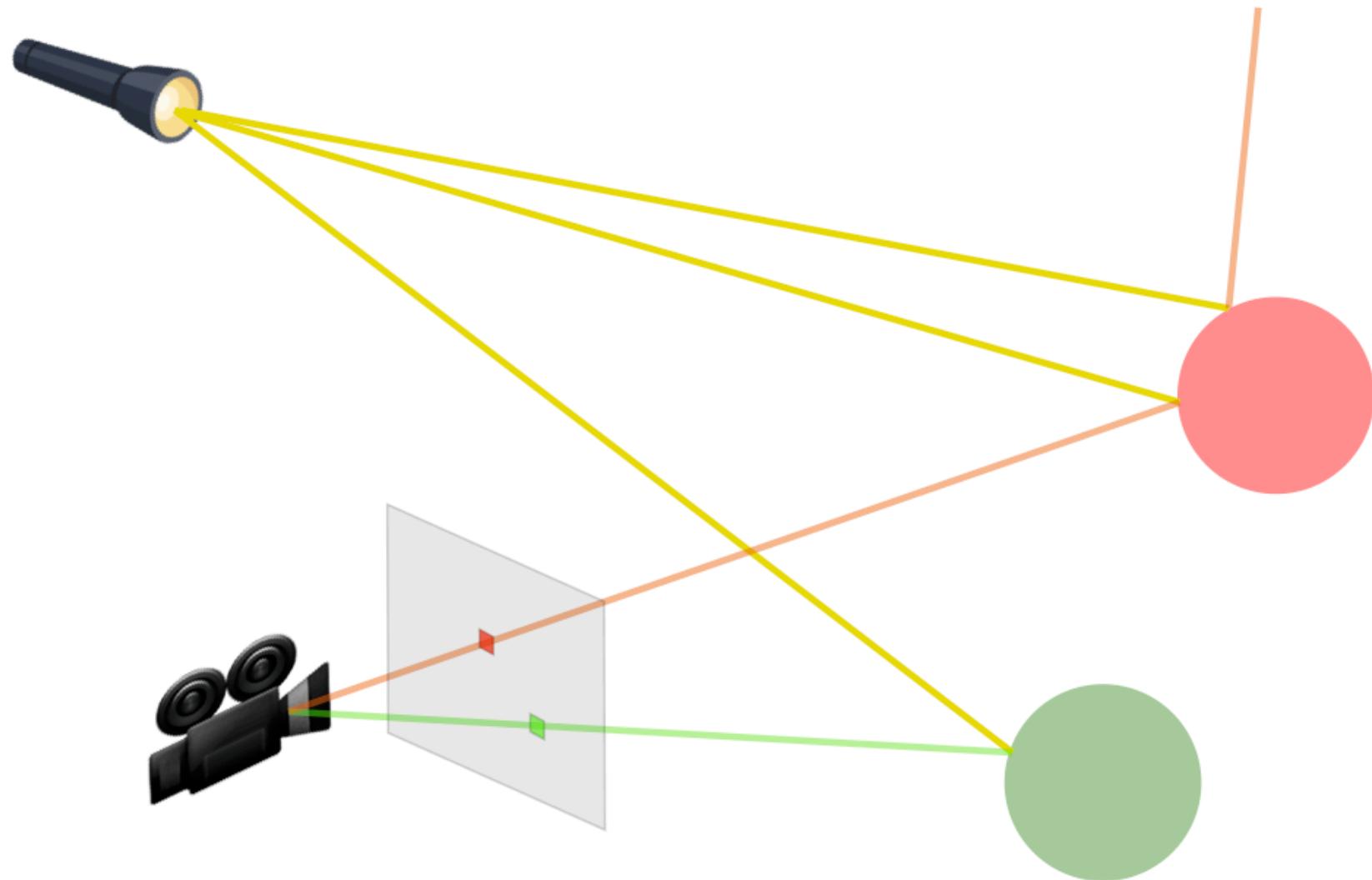
- ▶ World (spheres), light source, camera
- ▶ Incident rays
- ▶ Reflected rays
- ▶ Discarded rays
- ▶ Canvas

# RAY TRACING



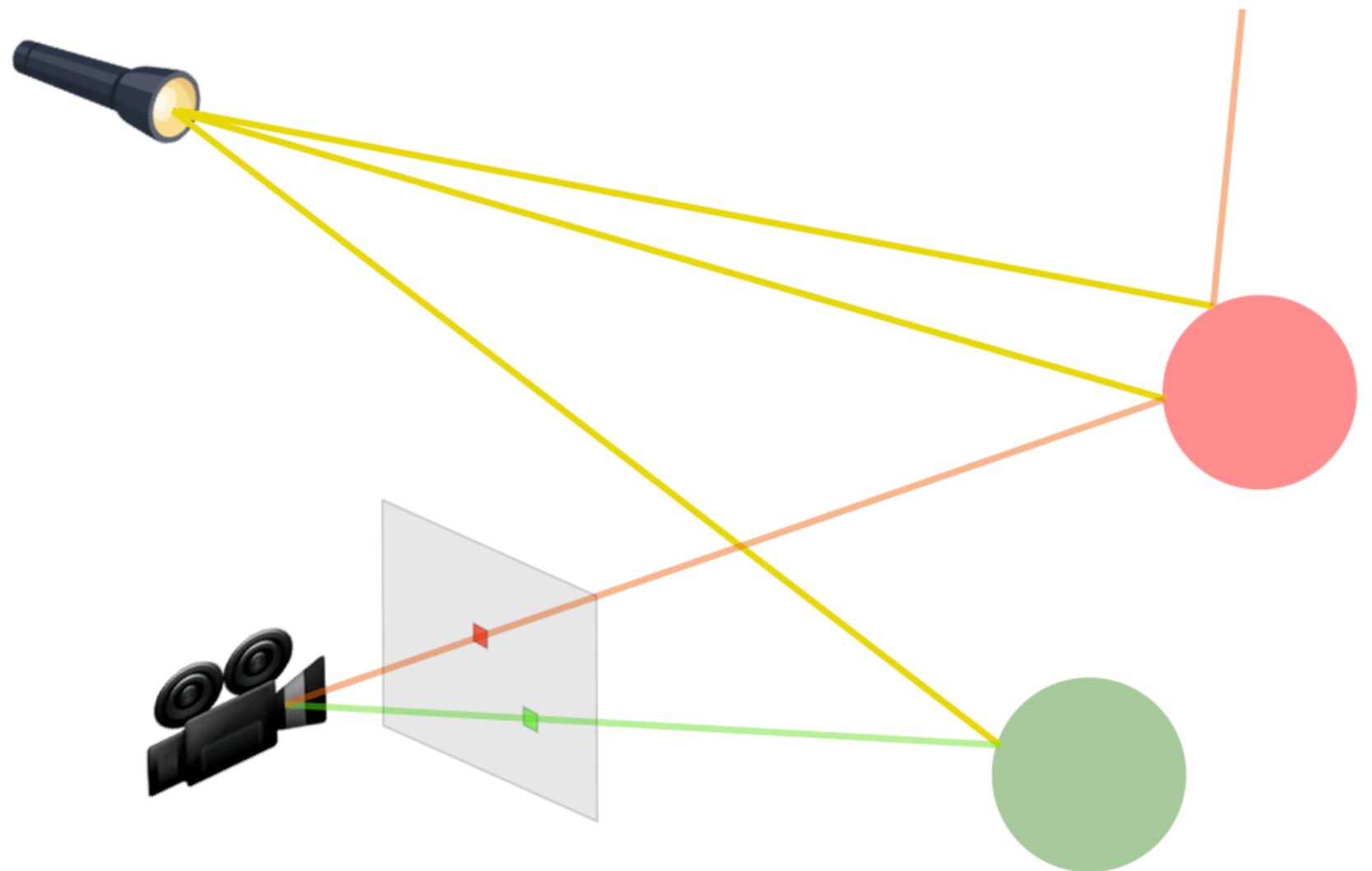
- ▶ World (spheres), light source, camera
- ▶ Incident rays
- ▶ Reflected rays
- ▶ Discarded rays
- ▶ Canvas
- ▶ Colored pixels

# RAY TRACING



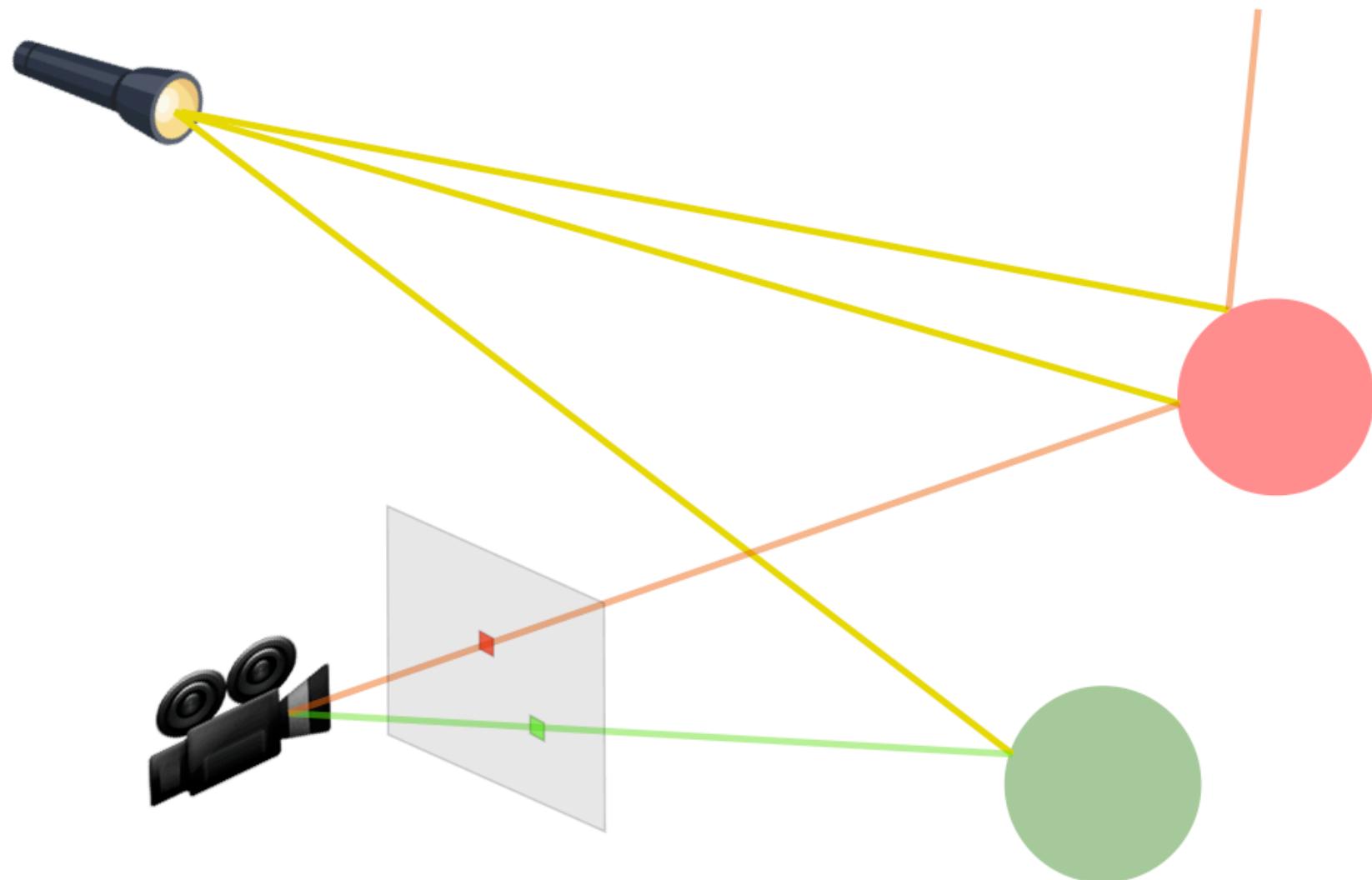
- ▶ World (spheres), light source, camera
- ▶ Incident rays
- ▶ Reflected rays
- ▶ Discarded rays
- ▶ Canvas
- ▶ Colored pixels

# RAY TRACING



**Options:**

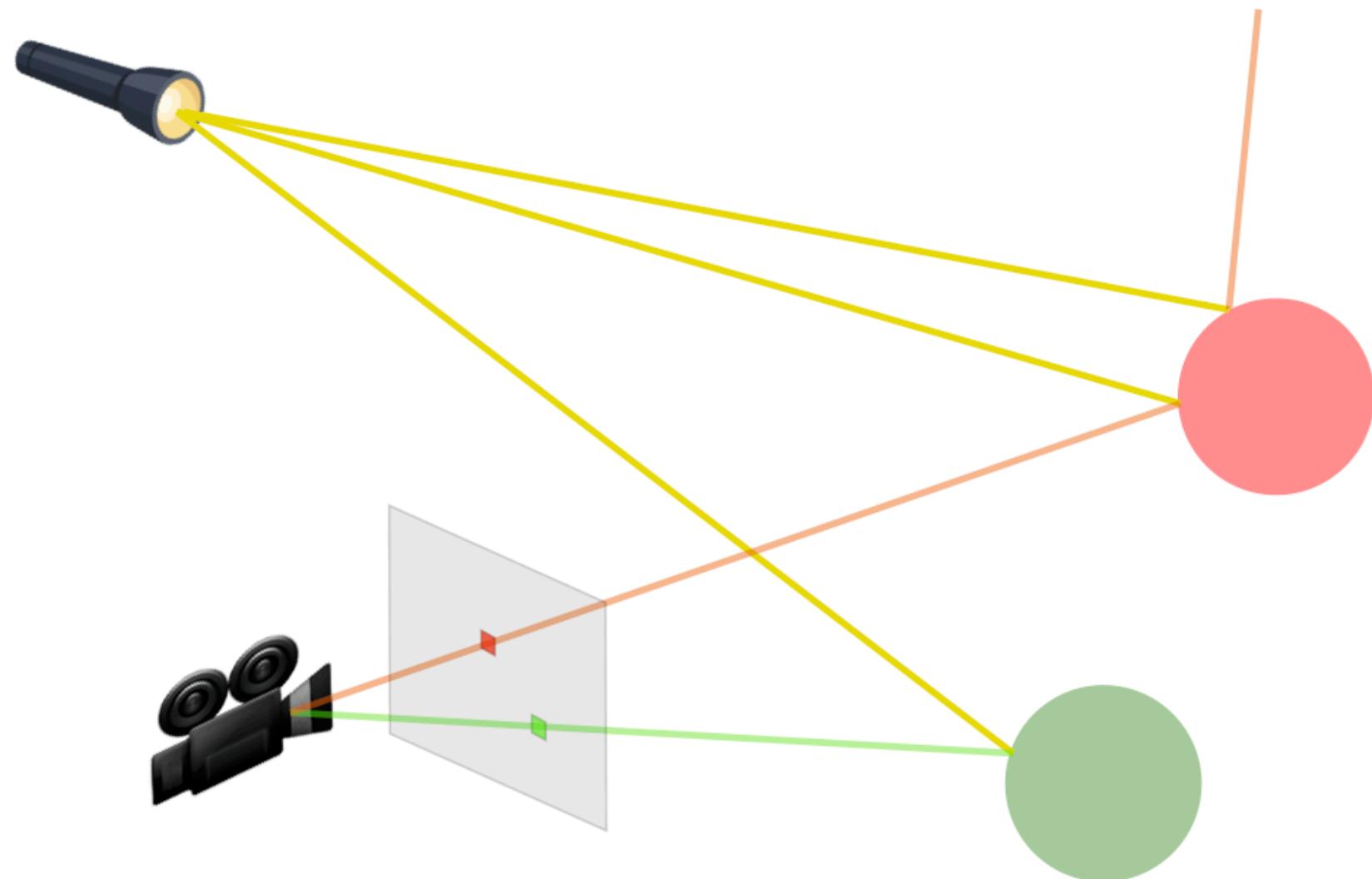
# RAY TRACING



**Options:**

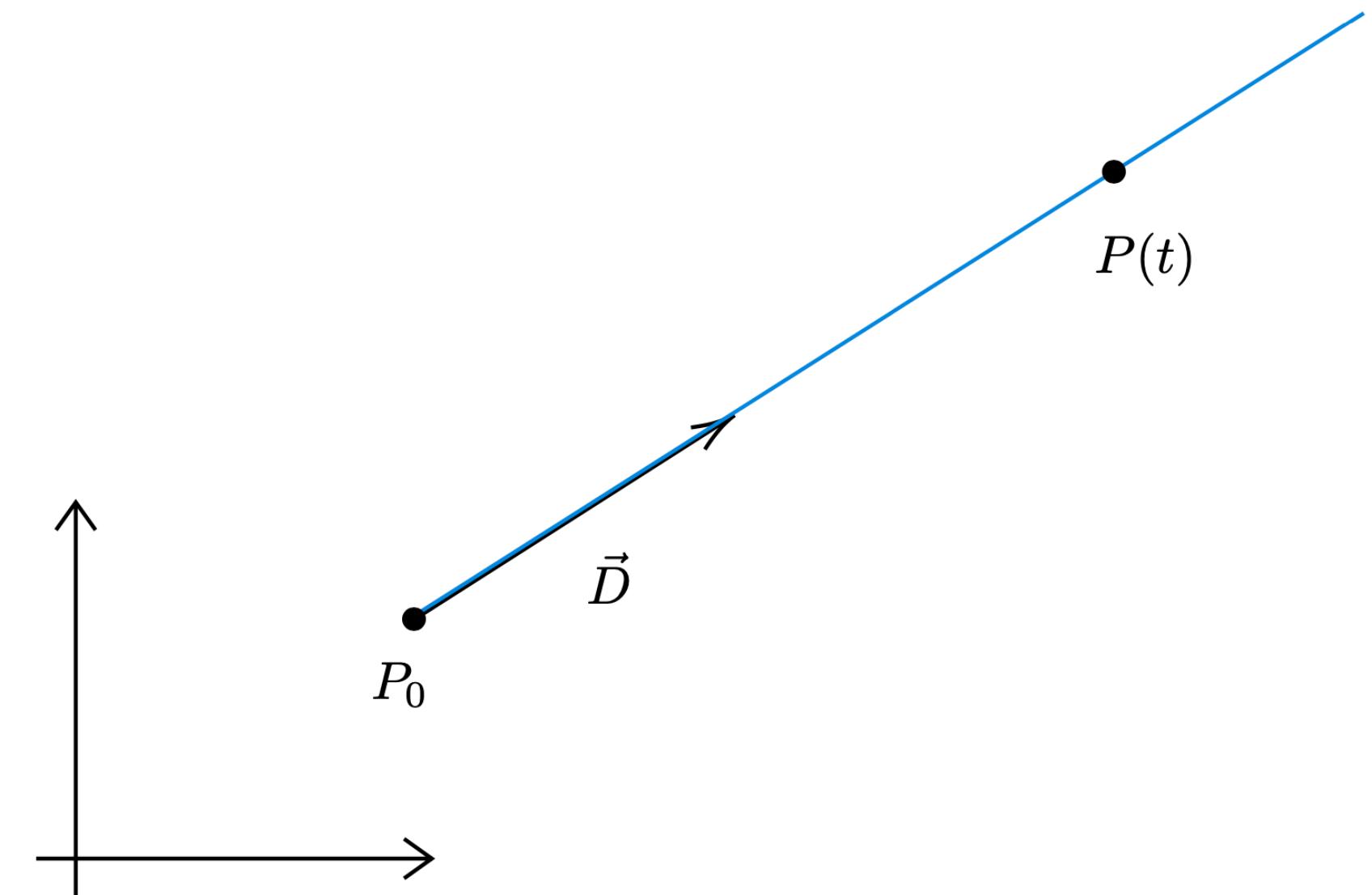
1. Compute all the rays (and discard most of them)

# 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



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

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

**RAY**

# TRANSFORMATIONS MODULE

```
trait AT

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

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

```
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
```

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

# 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
```

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

# 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
```

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

▶

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

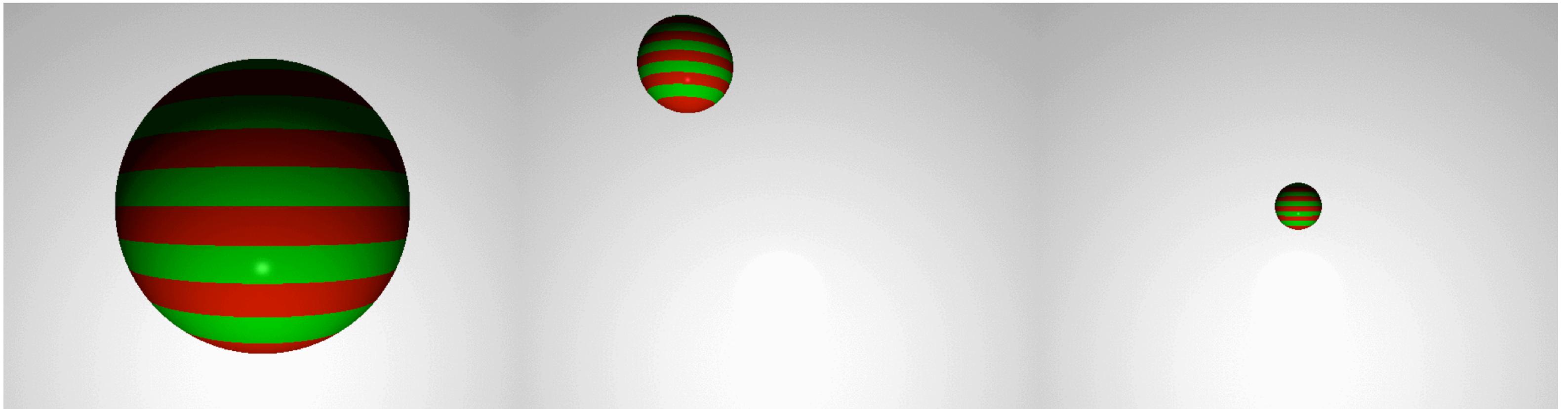
# TRANSFORMATIONS MODULE - LIVE

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

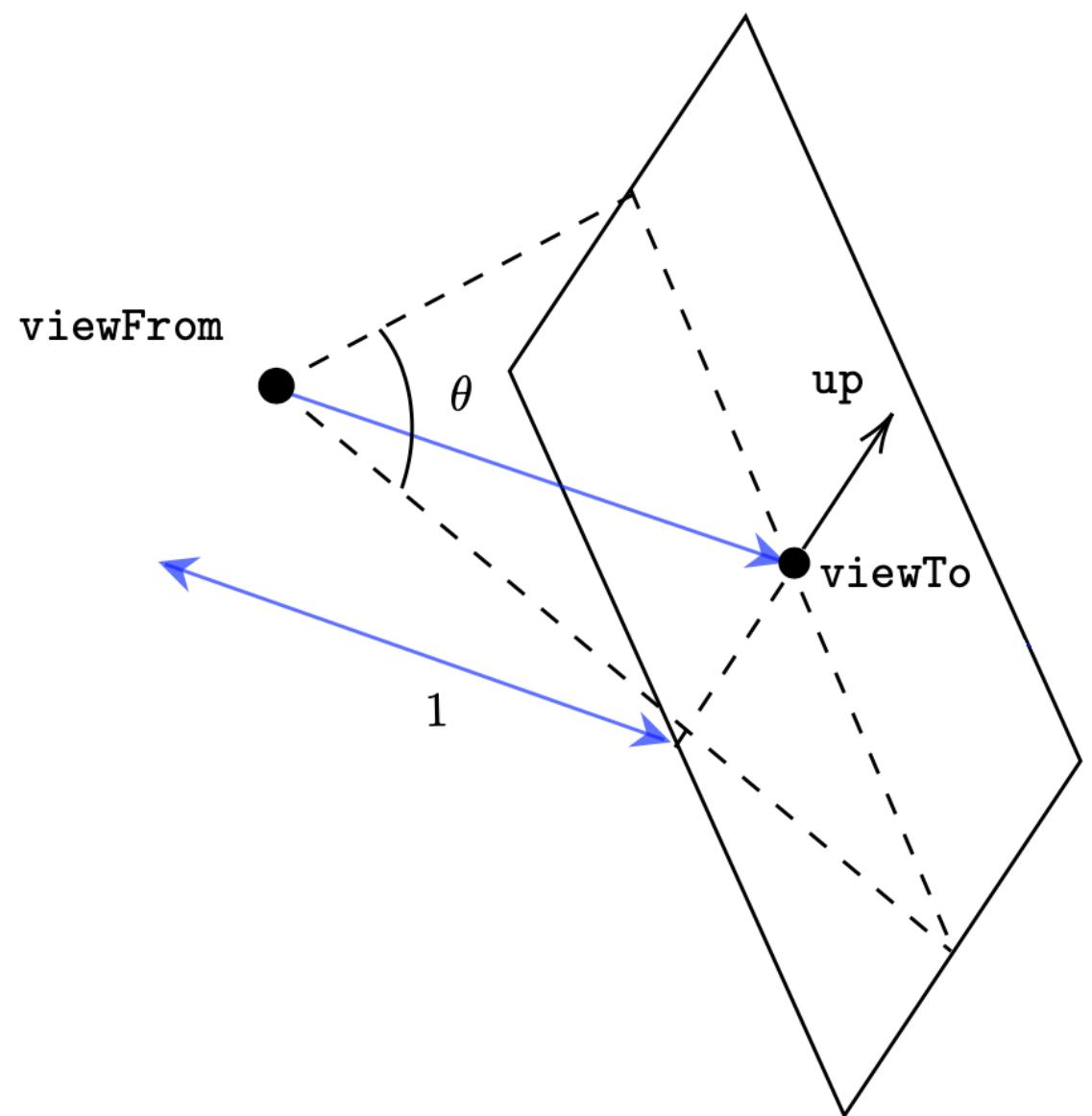
```
val live: ZLayer[MatrixModule, Nothing, ATModule] =
  ZLayer.fromService { matrixSvc =>
    new Service {
      def applyTf(tf: AT, vec: Vec) =
        matrixSvc.mul(tf.direct, v)

      /* ... */
    }
  }
```

# LAYER 1: TRANSFORMATIONS

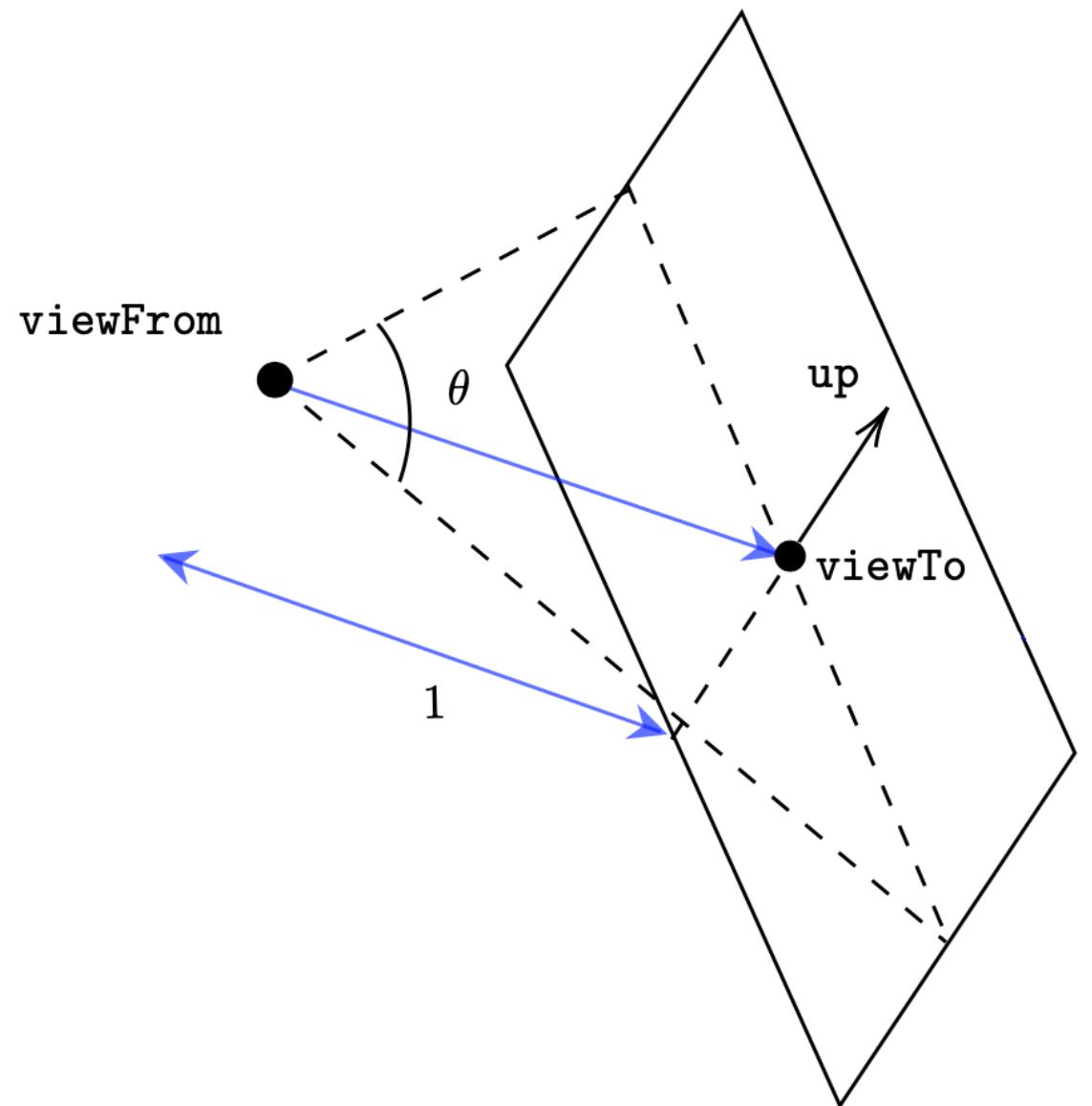


# CAMERA



```
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



```
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  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$
- ▶ 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  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$
- ▶ 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  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$
- ▶ 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  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$
- ▶ 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  $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$
- ▶ 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])
```

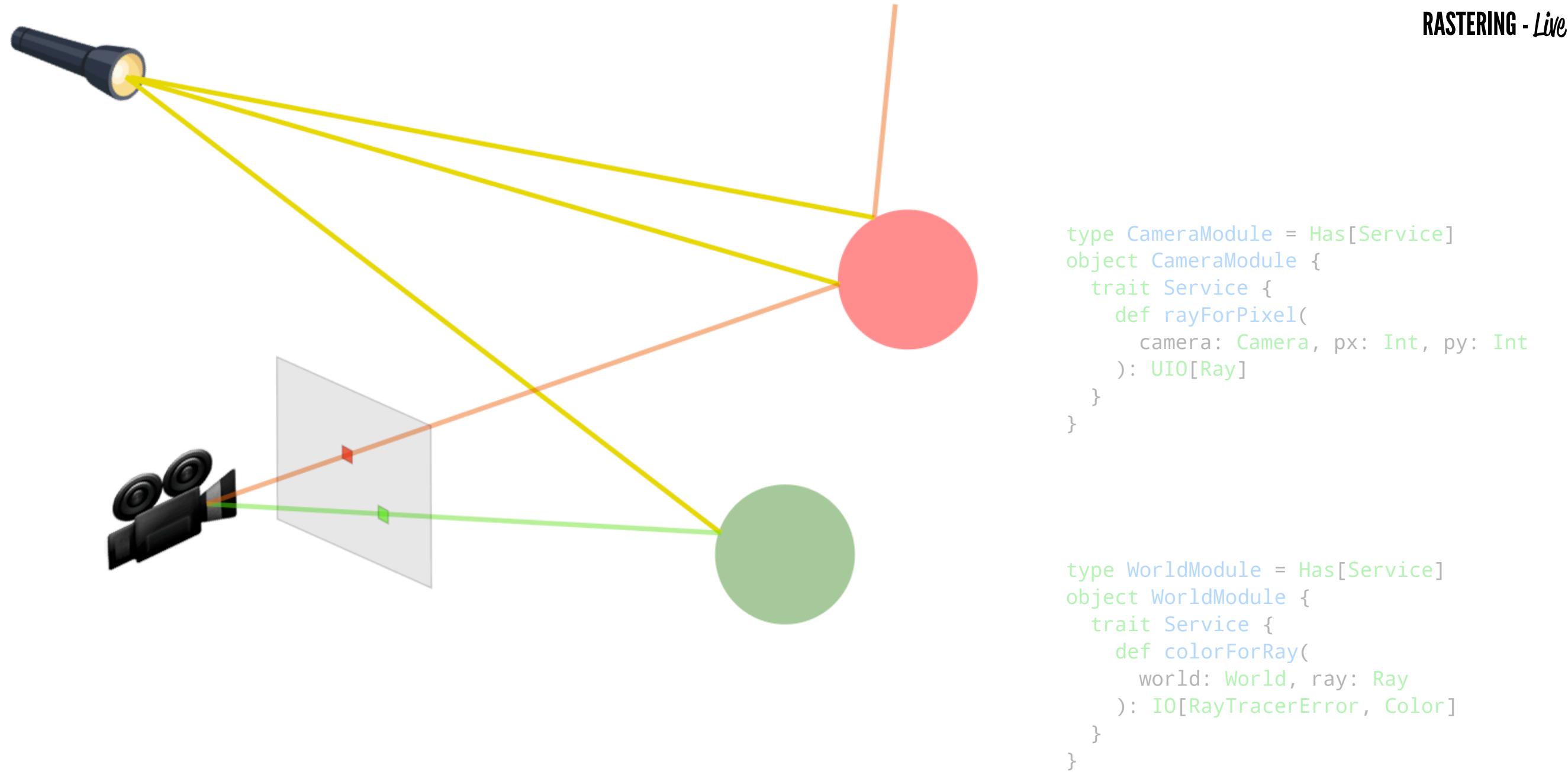
- ▶ Everything requires ATModule

# WORLD RENDERING - TOP DOWN

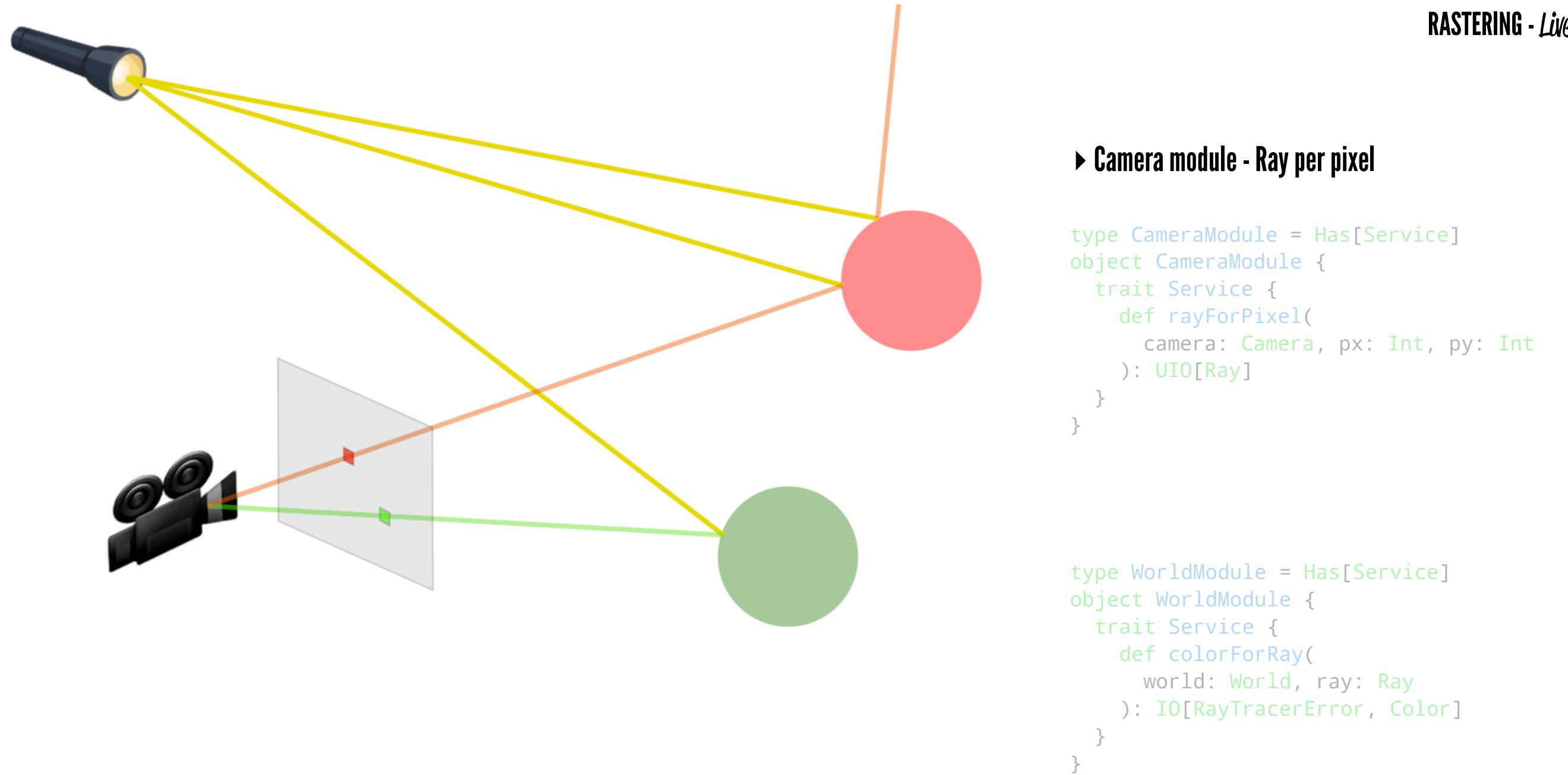
## RASTERING - GENERATE A STREAM OF COLORED PIXELS

```
type RasteringModule = Has[Service]
object RasteringModule {
  trait Service {
    def raster(world: World, camera: Camera): Stream[RayTracerError, ColoredPixel]
  }
}
```

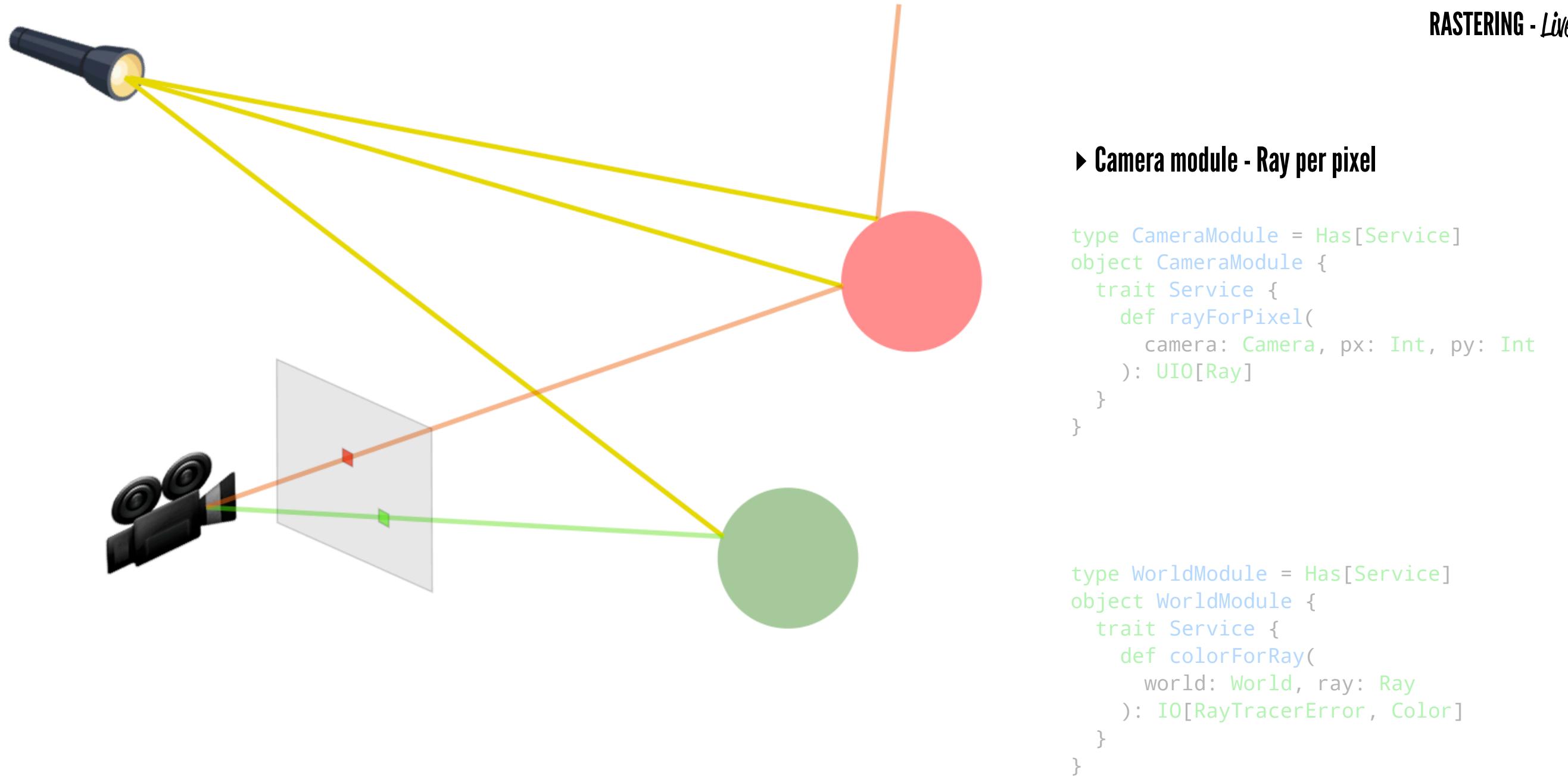
# WORLD RENDERING - TOP DOWN



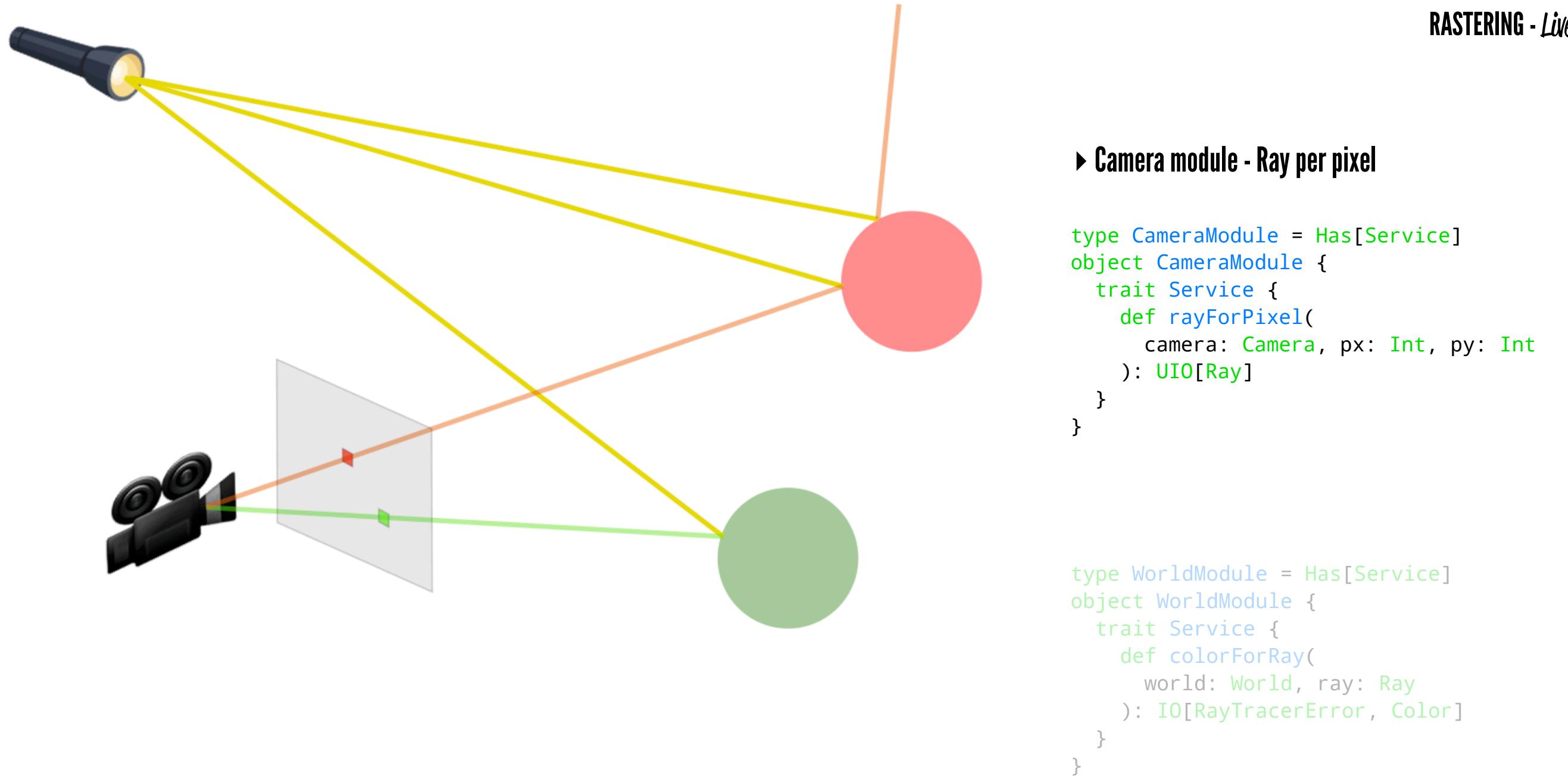
# WORLD RENDERING - TOP DOWN



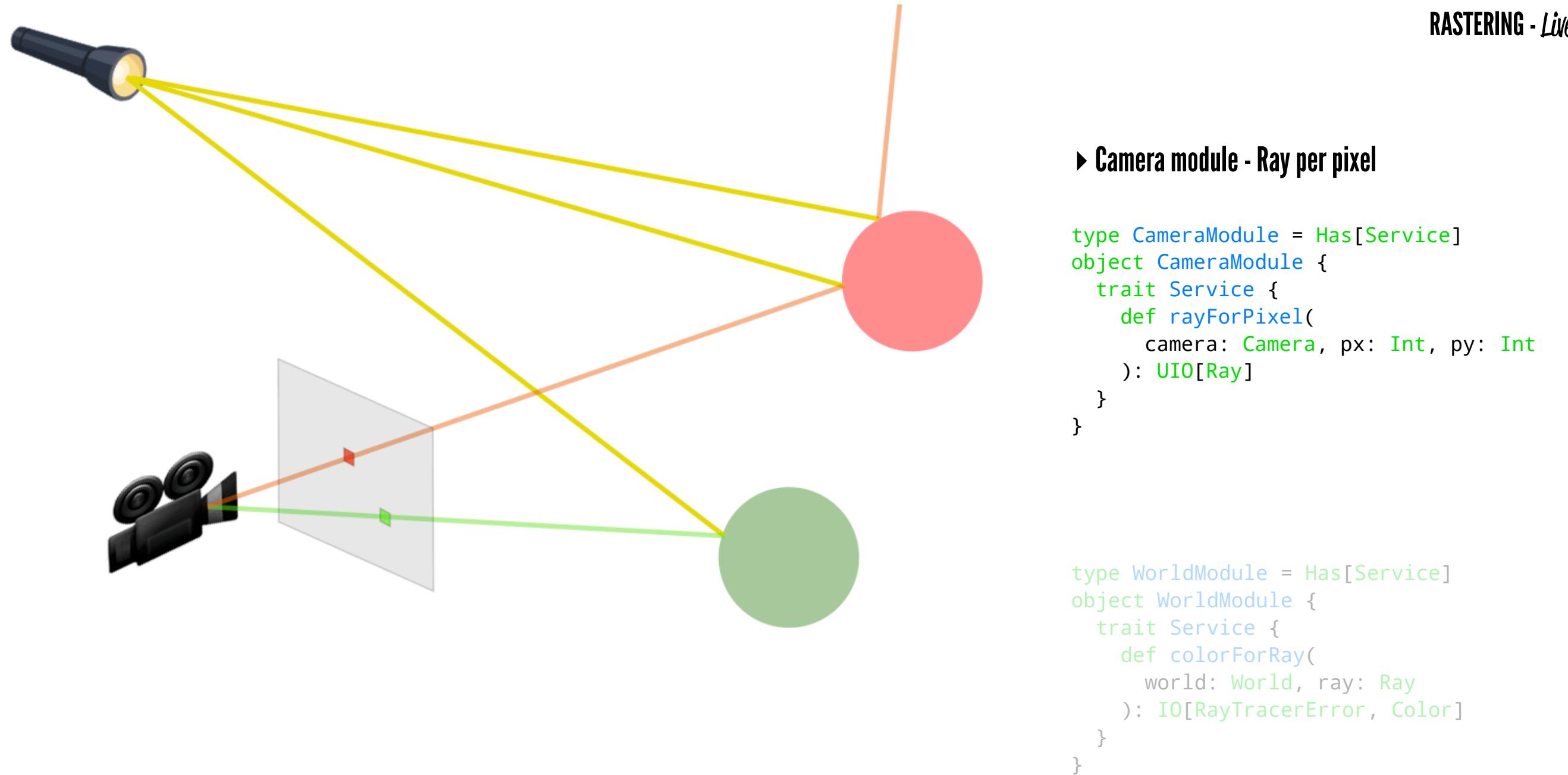
# WORLD RENDERING - TOP DOWN



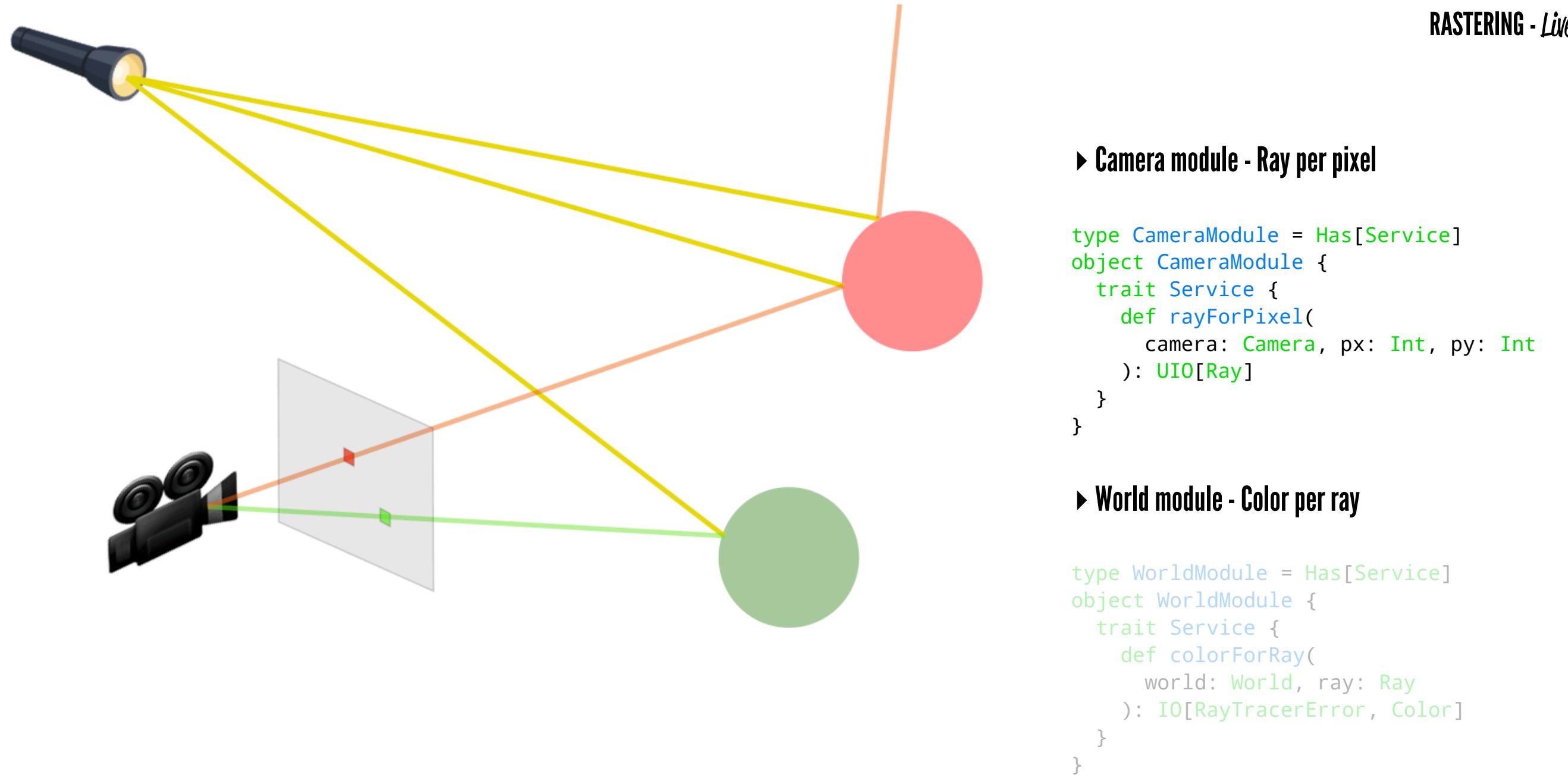
# WORLD RENDERING - TOP DOWN



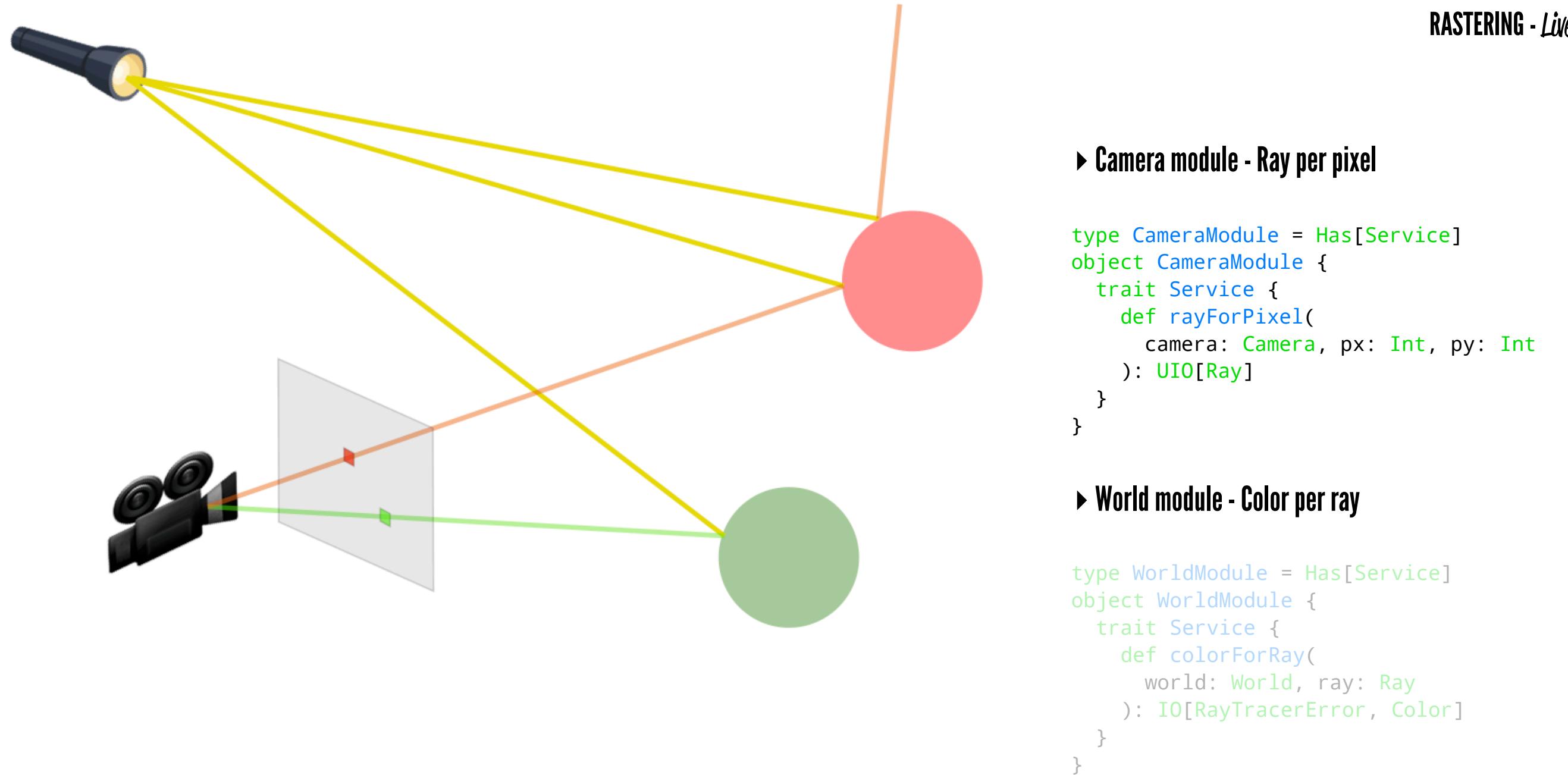
# WORLD RENDERING - TOP DOWN



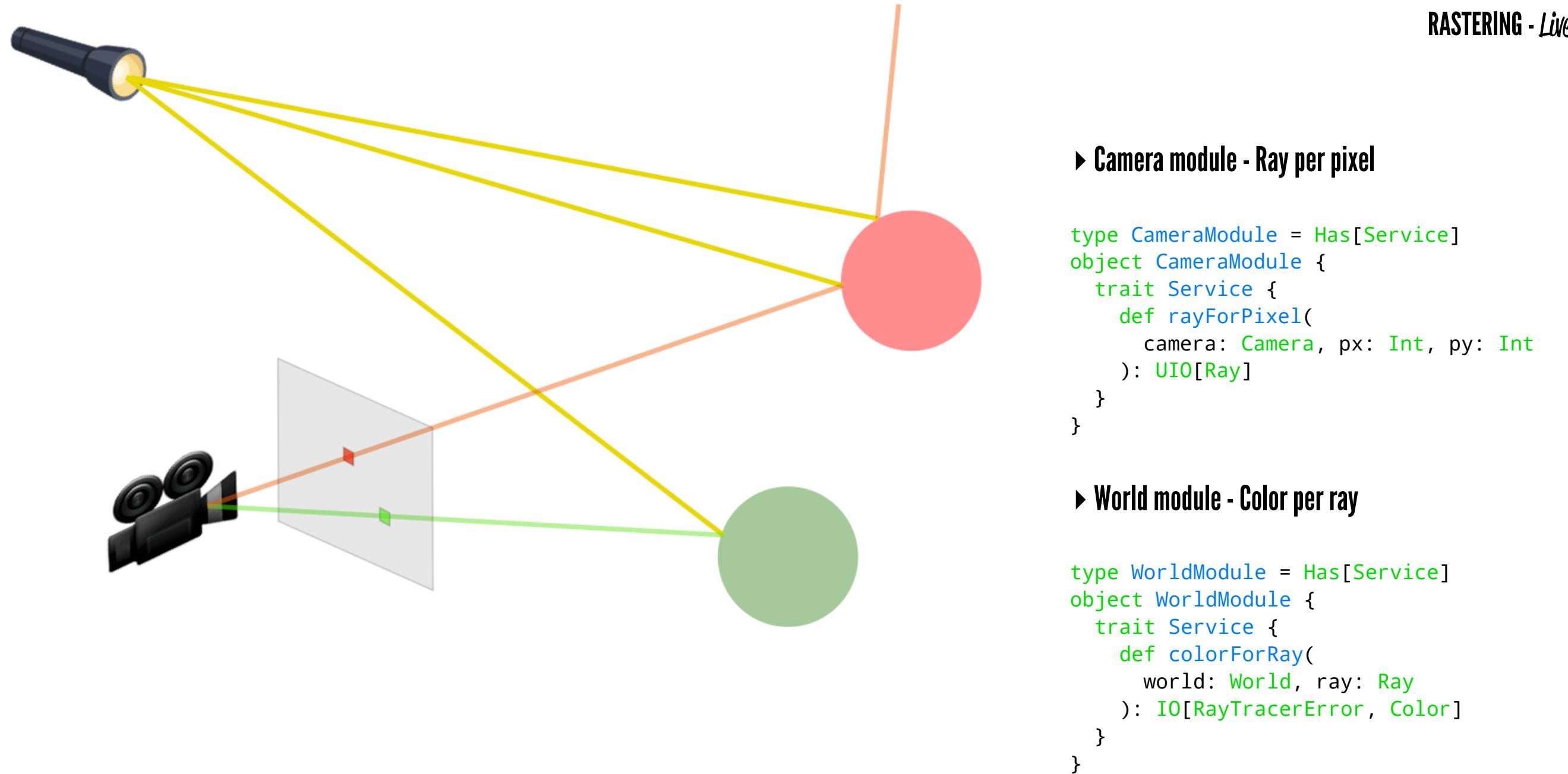
# WORLD RENDERING - TOP DOWN



# WORLD RENDERING - TOP DOWN



# WORLD RENDERING - TOP DOWN



# WORLD RENDERING - TOP DOWN

## RASTERING *Live* - MODULE DEPENDENCY

```
val chunkRasteringModule: ZLayer[CameraModule with WorldModule, Nothing, RasteringModule] =  
  ZLayer.fromServices[cameraModule.Service, worldModule.Service, rasteringModule.Service] {  
    (cameraSvc, worldSvc) =>  
      new Service {  
        override def raster(world: World, camera: Camera):  
          Stream[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

## RASTERING *Live* - MODULE DEPENDENCY

```
val chunkRasteringModule: ZLayer[CameraModule with WorldModule, Nothing, RasteringModule] =  
  ZLayer.fromServices[cameraModule.Service, worldModule.Service, rasteringModule.Service] {  
    (cameraSvc, worldSvc) =>  
      new Service {  
        override def raster(world: World, camera: Camera):  
          Stream[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

## RASTERING *Live* - MODULE DEPENDENCY

```
val chunkRasteringModule: ZLayer[CameraModule with WorldModule, Nothing, RasteringModule] =  
  ZLayer.fromServices[cameraModule.Service, worldModule.Service, rasteringModule.Service] {  
    (cameraSvc, worldSvc) =>  
      new Service {  
        override def raster(world: World, camera: Camera):  
          Stream[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)  
            }  
          }  
      }  
  }
```

# **LAYERS**

## **TAKEAWAY**

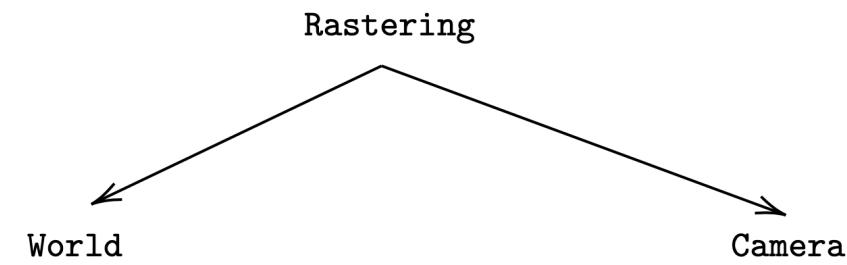
**IMPLEMENT AND TEST EVERY LAYER ONLY IN TERMS OF THE IMMEDIATELY UNDERLYING LAYER**

**IT'S MODULES**



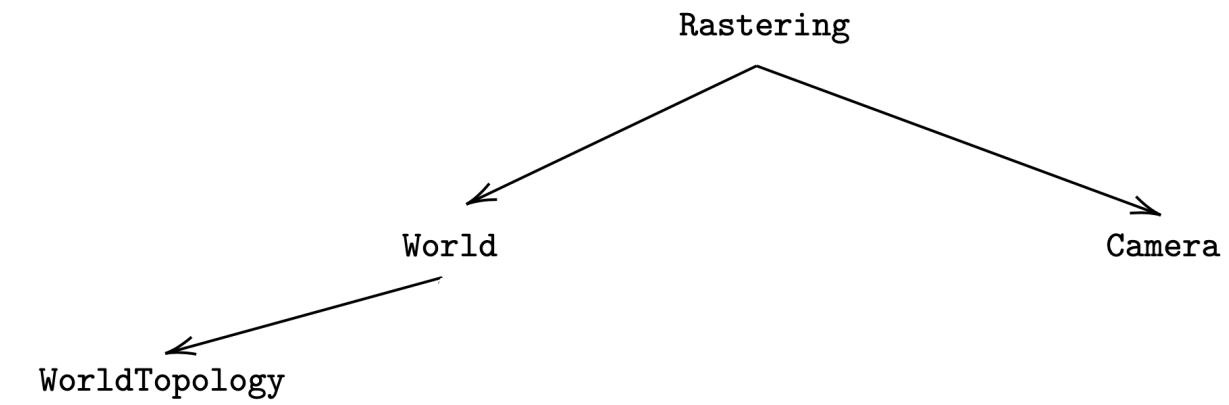
**ALL THE WAY DOWN**

## LIVE CameraModule



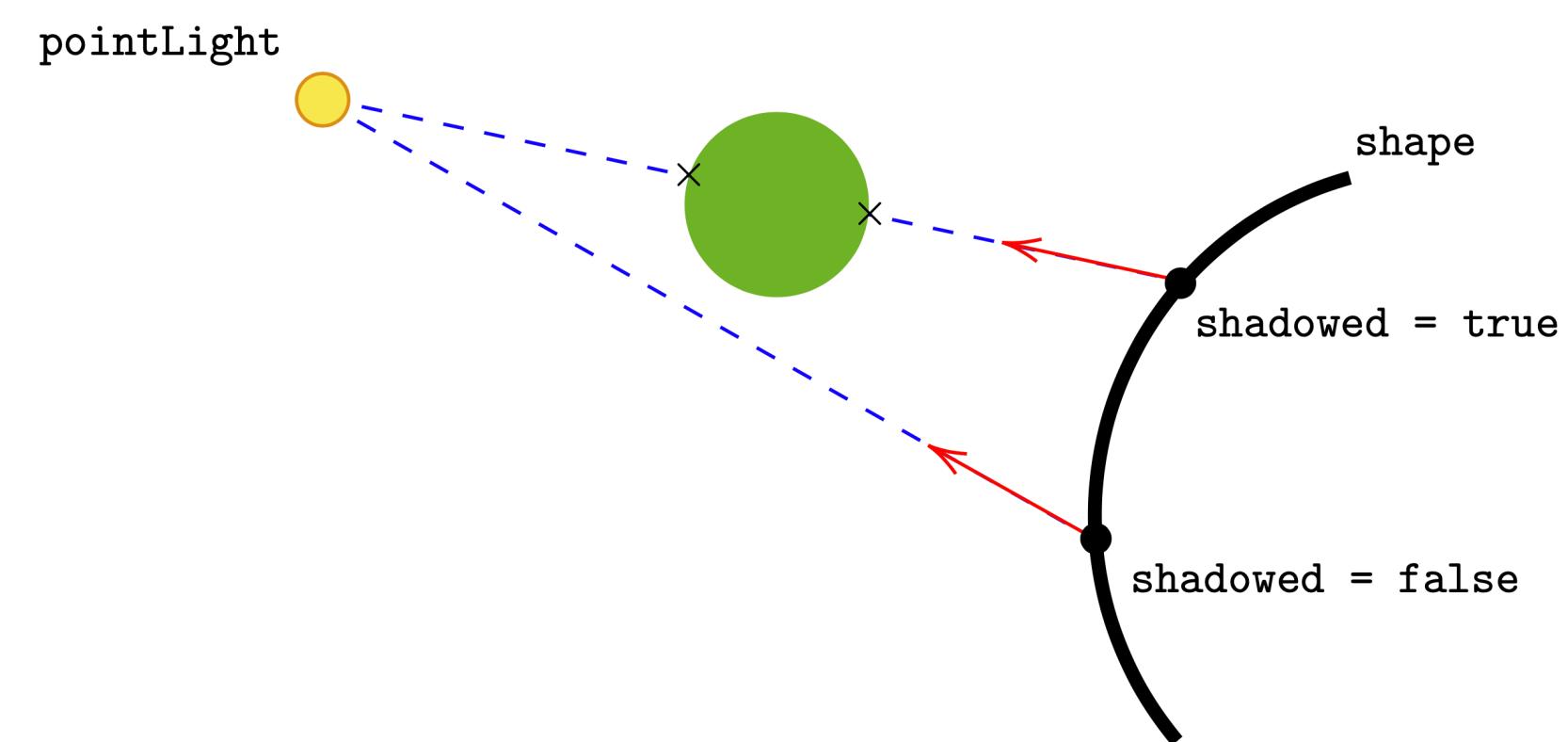
```
object CameraModule {  
    val live: ZLayer[ATModule, Nothing, CameraModule] =  
        ZLayer.fromService { atSvc =>  
            /* ... */  
        }  
}
```

# LIVE WorldModule



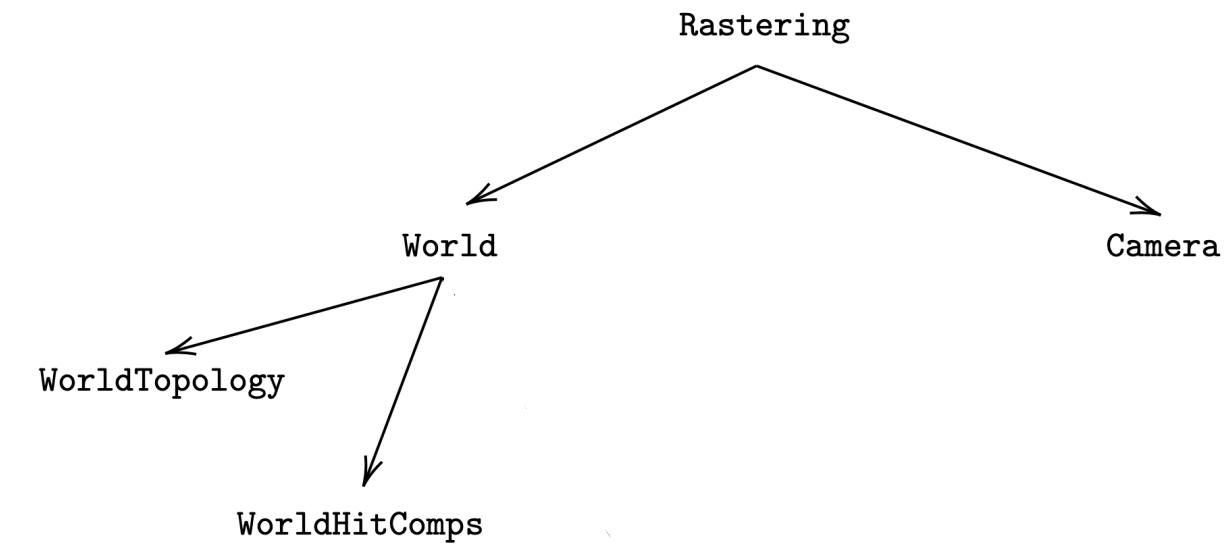
```
object WorldTopologyModule {
    trait Service {
        def intersections(world: World, ray: Ray): UIO[List[Intersection]]
        def isShadowed(world: World, pt: Pt): UIO[Boolean]
    }
}
```

# LIVE WorldModule



```
WorldTopologyModule
object WorldTopologyModule {
  trait Service {
    def intersections(world: World, ray: Ray): UIO[List[Intersection]]
    def isShadowed(world: World, pt: Pt): UIO[Boolean]
  }
}
```

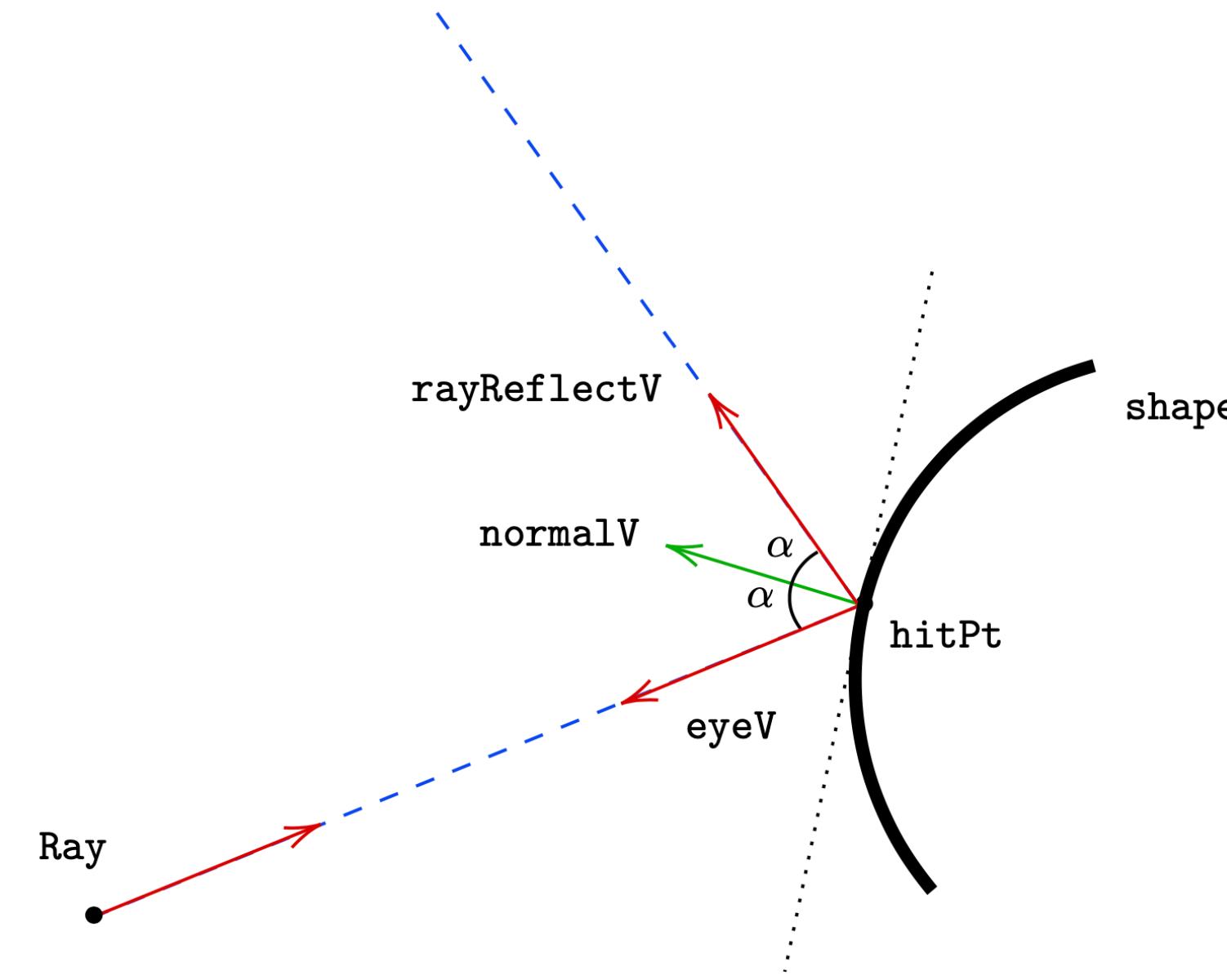
# LIVE WorldModule



*WorldHitCompsModule*

```
case class HitComps(  
    shape: Shape, hitPt: Pt, normalV: Vec, eyeV: Vec,  
    rayReflectV: Vec, n1: Double = 1, n2: Double = 1  
)  
  
object WorldHitCompsModule {  
    trait Service {  
        def hitComps(  
            ray: Ray, hit: Intersection,  
            intersections: List[Intersection]  
        ): IO[GenericError, HitComps]  
    }  
}
```

# LIVE WorldModule



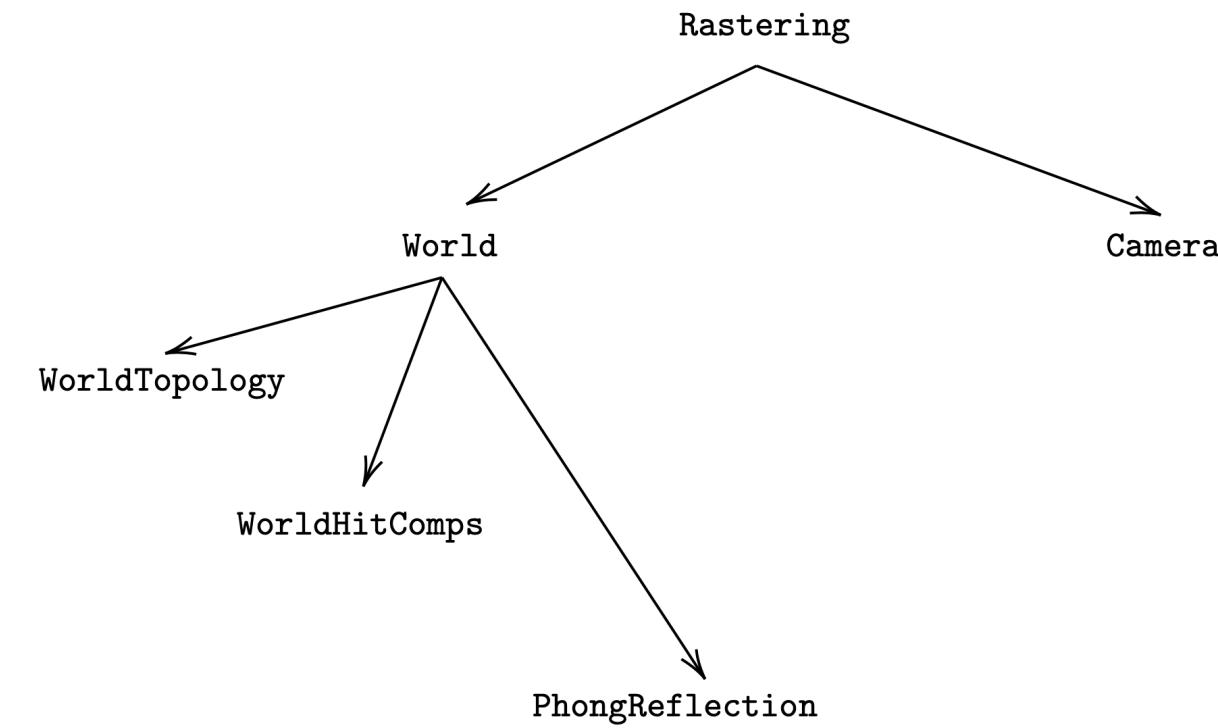
HitComps(shape, hitPt, normalV, eyeV, rayReflectV)

```
WorldHitCompsModule

case class HitComps(
    shape: Shape, hitPt: Pt, normalV: Vec, eyeV: Vec,
    rayReflectV: Vec, n1: Double = 1, n2: Double = 1
)

object WorldHitCompsModule {
    trait Service {
        def hitComps(
            ray: Ray, hit: Intersection,
            intersections: List[Intersection]
        ): I0[GenericError, HitComps]
    }
}
```

# LIVE WorldModule

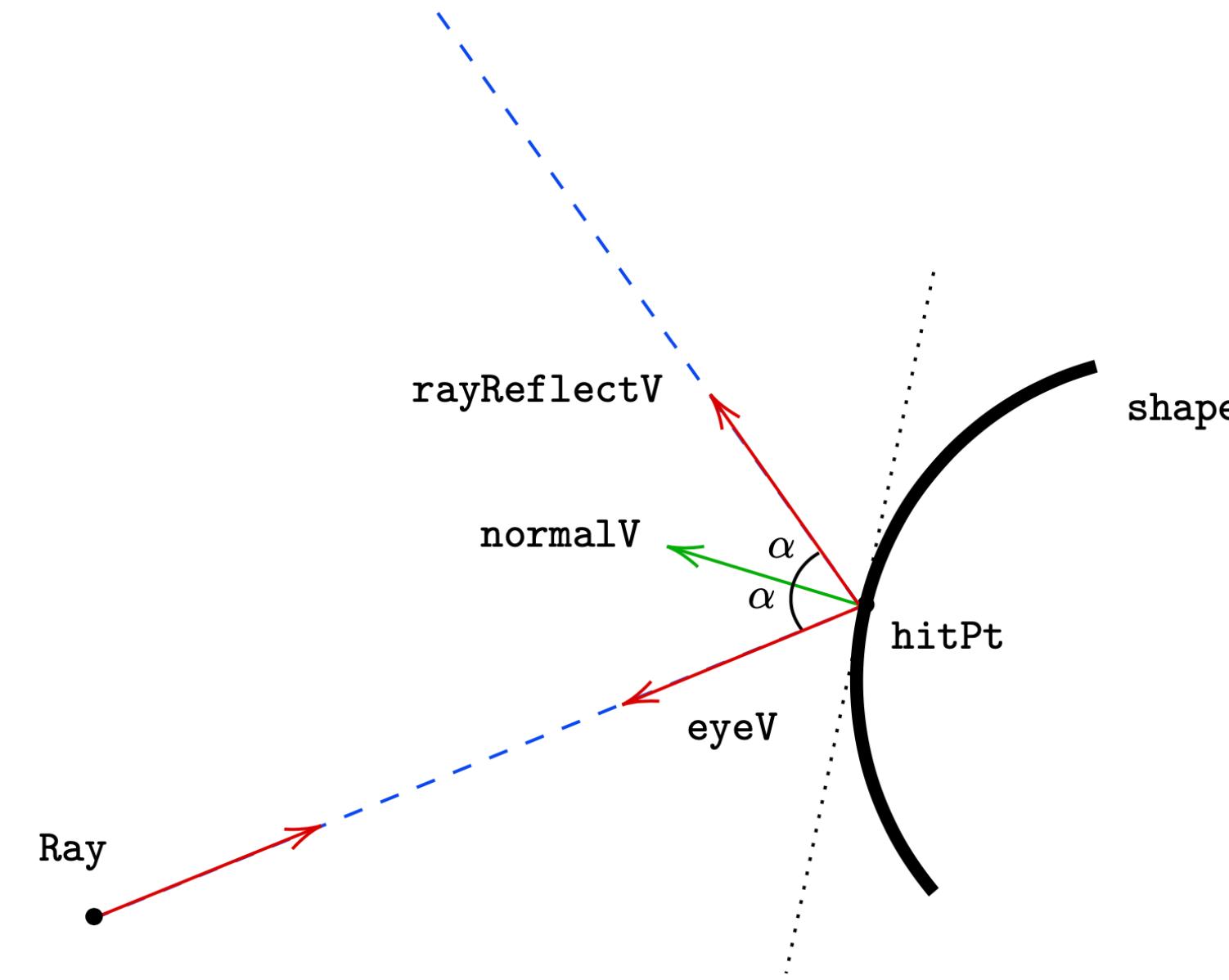


```
PhongReflectionModule

case class PhongComponents(
    ambient: Color, diffuse: Color, reflective: Color
) {
    def toColor: Color = ambient + diffuse + reflective
}

object PhongReflectionModule {
    trait Service {
        def lighting(
            pointLight: PointLight, hitComps: HitComps,
            inShadow: Boolean
        ): UIO[PhongComponents]
    }
}
```

# LIVE WorldModule



HitComps(shape, hitPt, normalV, eyeV, rayReflectV)

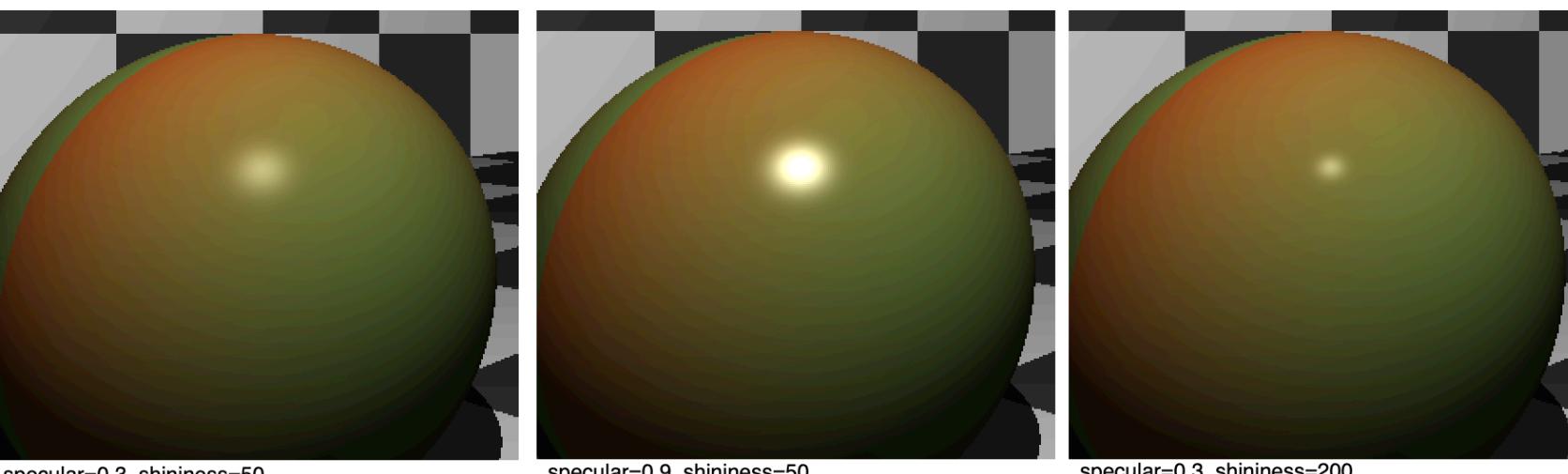
```
PhongReflectionModule

case class PhongComponents(
    ambient: Color, diffuse: Color, reflective: Color
) {
    def toColor: Color = ambient + diffuse + reflective
}

object PhongReflectionModule {
    trait Service {
        def lighting(
            pointLight: PointLight, hitComps: HitComps,
            inShadow: Boolean
        ): UIO[PhongComponents]
    }
}
```

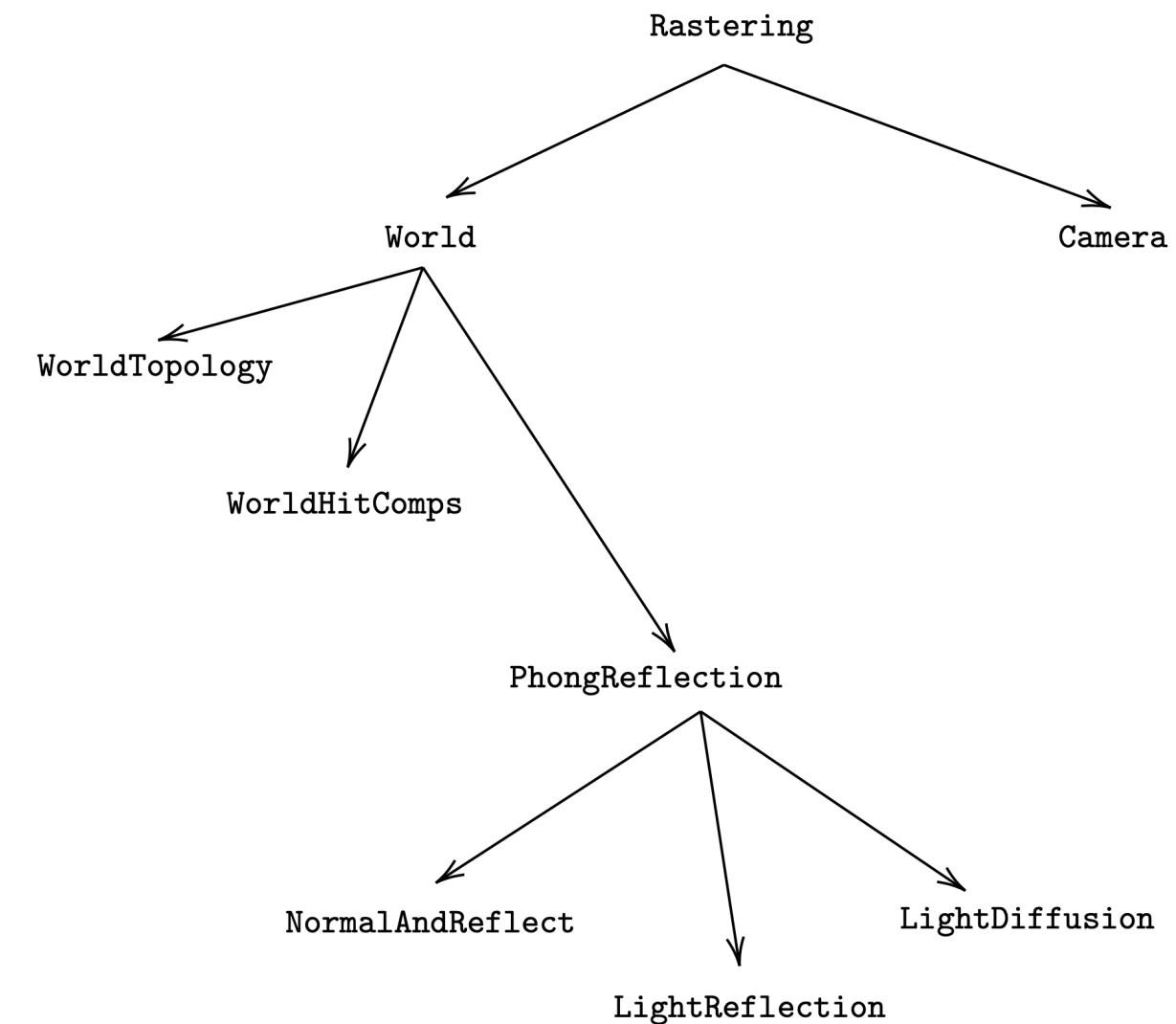
# REFLECT THE LIGHT SOURCE

## 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

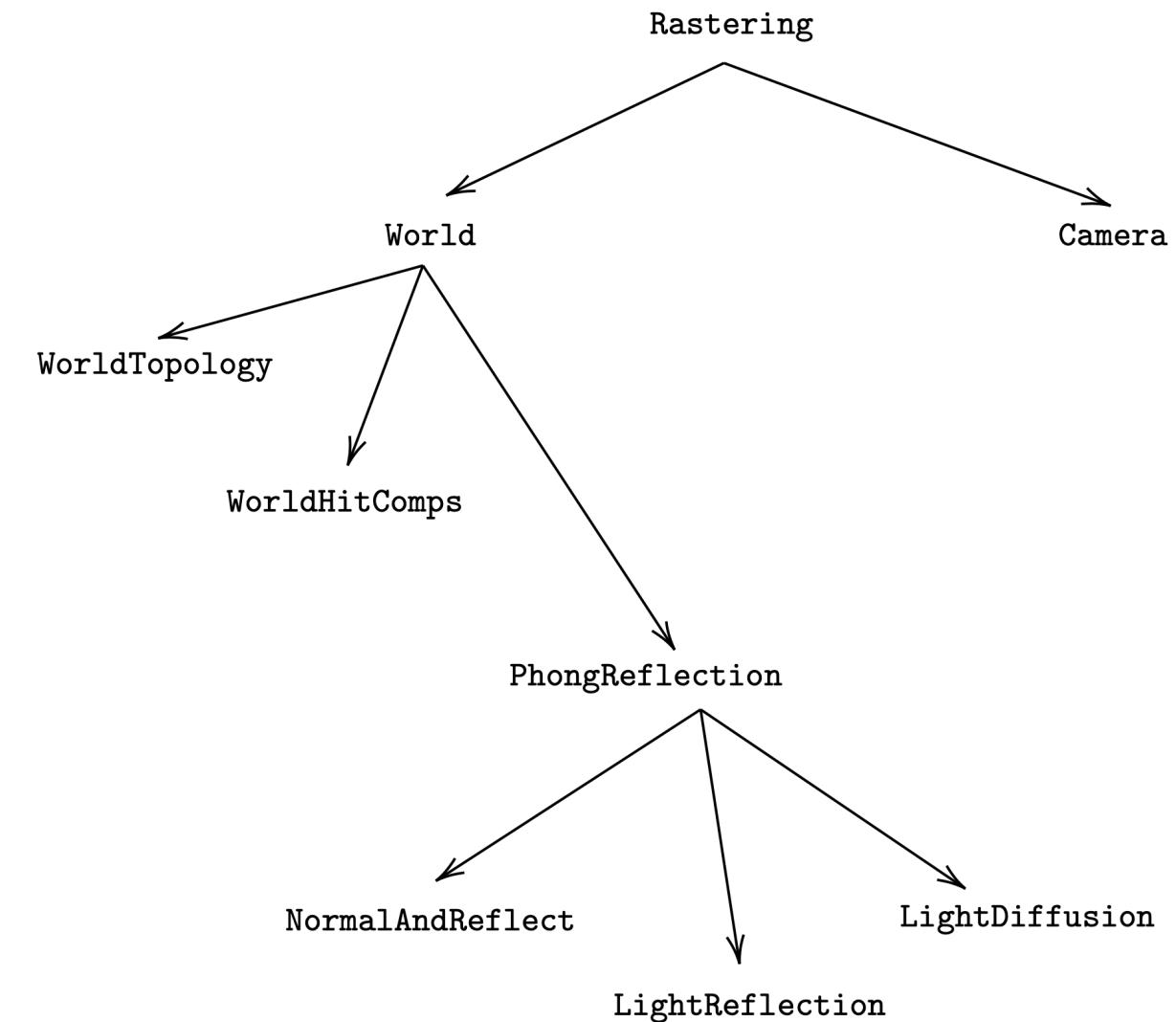


WITH *LightDiffusion AND LightReflection*

```
object PhongReflectionModule {
    trait Service { }

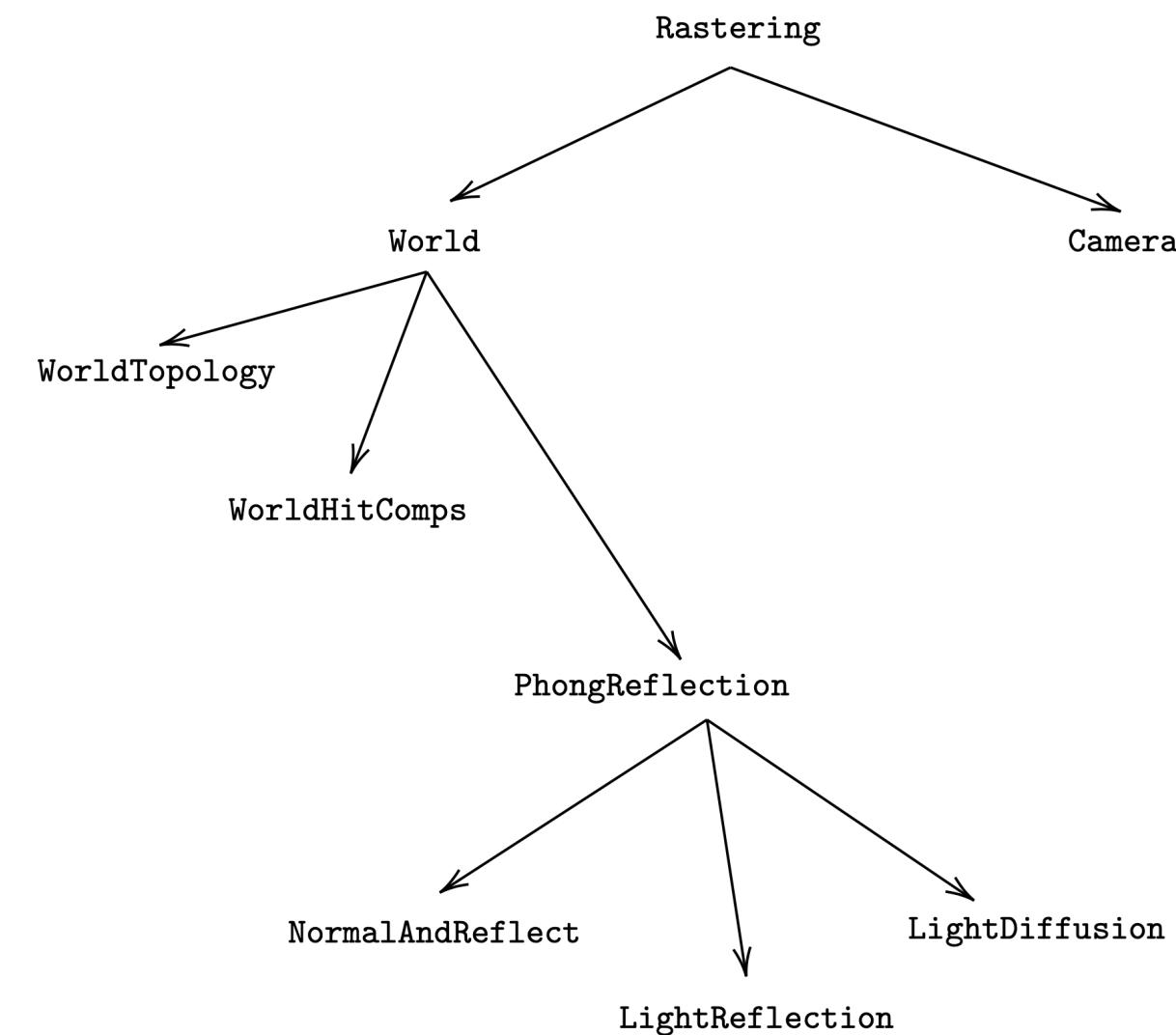
    val live: ZLayer[ATModule
        with LightDiffusionModule
        with LightReflectionModule,
        Nothing,
        PhongReflectionModule]
}
```

# DRAWING PROGRAM



```
def draw(sceneBundle: SceneBundle):  
  ZIO[CanvasSerializer  
    with RasteringModule  
    with ATModule,  
    Nothing,  
    Array[Byte]]
```

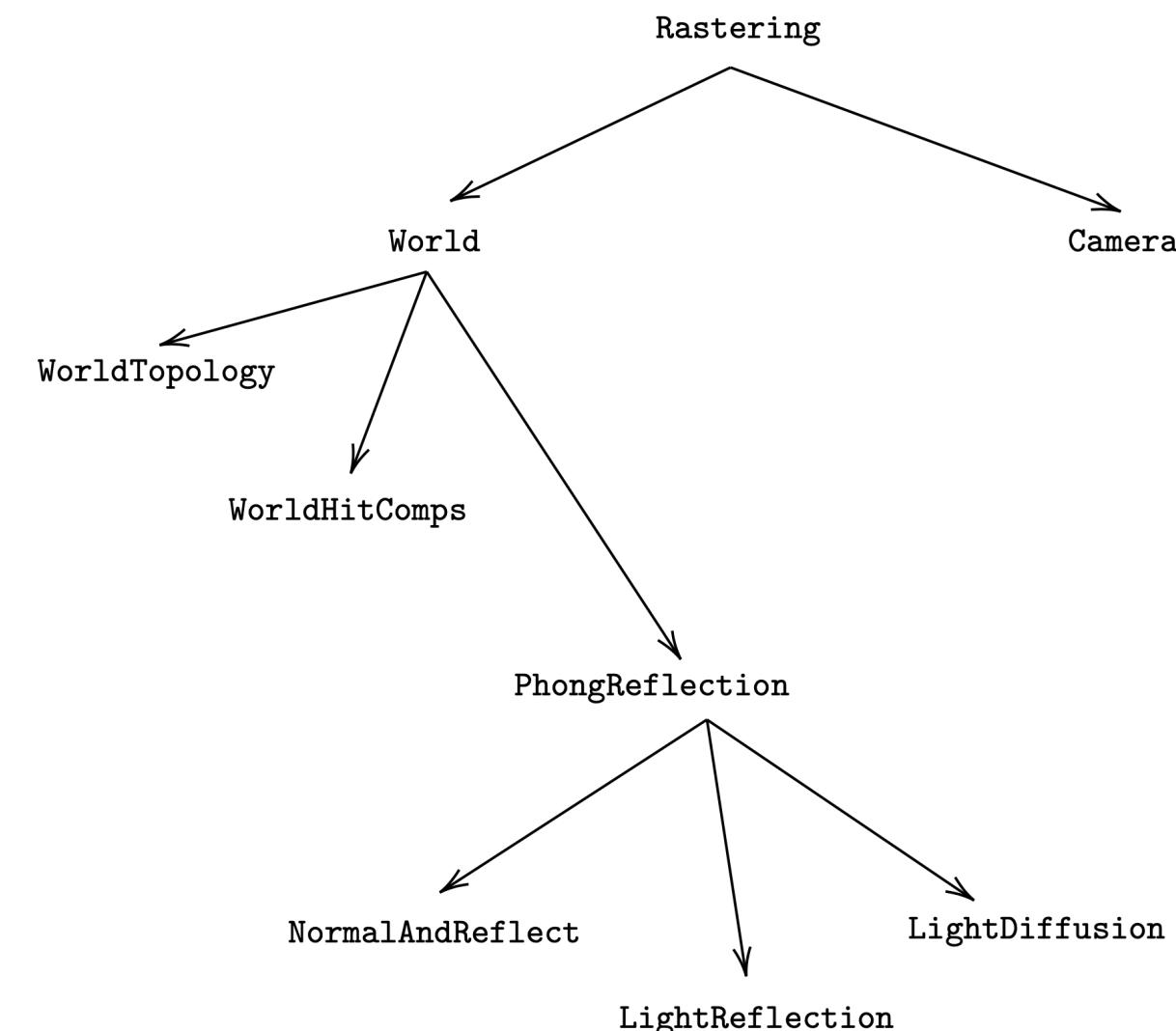
# WITH HTTP4S



```
class DrawRoutes[R <: CanvasSerializer with RasteringModule with ATModule] {  
    type F[A] = RIO[R, A]  
    private val http4sDsl = new Http4sDsl[F] {}  
    import http4sDsl._  
  
    val httpRoutes: HttpRoutes[F] = HttpRoutes.of[F] {  
        case req @ POST -> Root / "draw" =>  
            req.decode[Scene] { scene =>  
                (for {  
                    bundle   <- Http2World.httpScene2World(scene)  
                    bytes    <- draw(bundle)  
                } yield bytes).foldM {  
                    e => InternalServerError(s"something went wrong"),  
                    Ok(bytes, "image/png")  
                }  
            }  
    }  
}
```

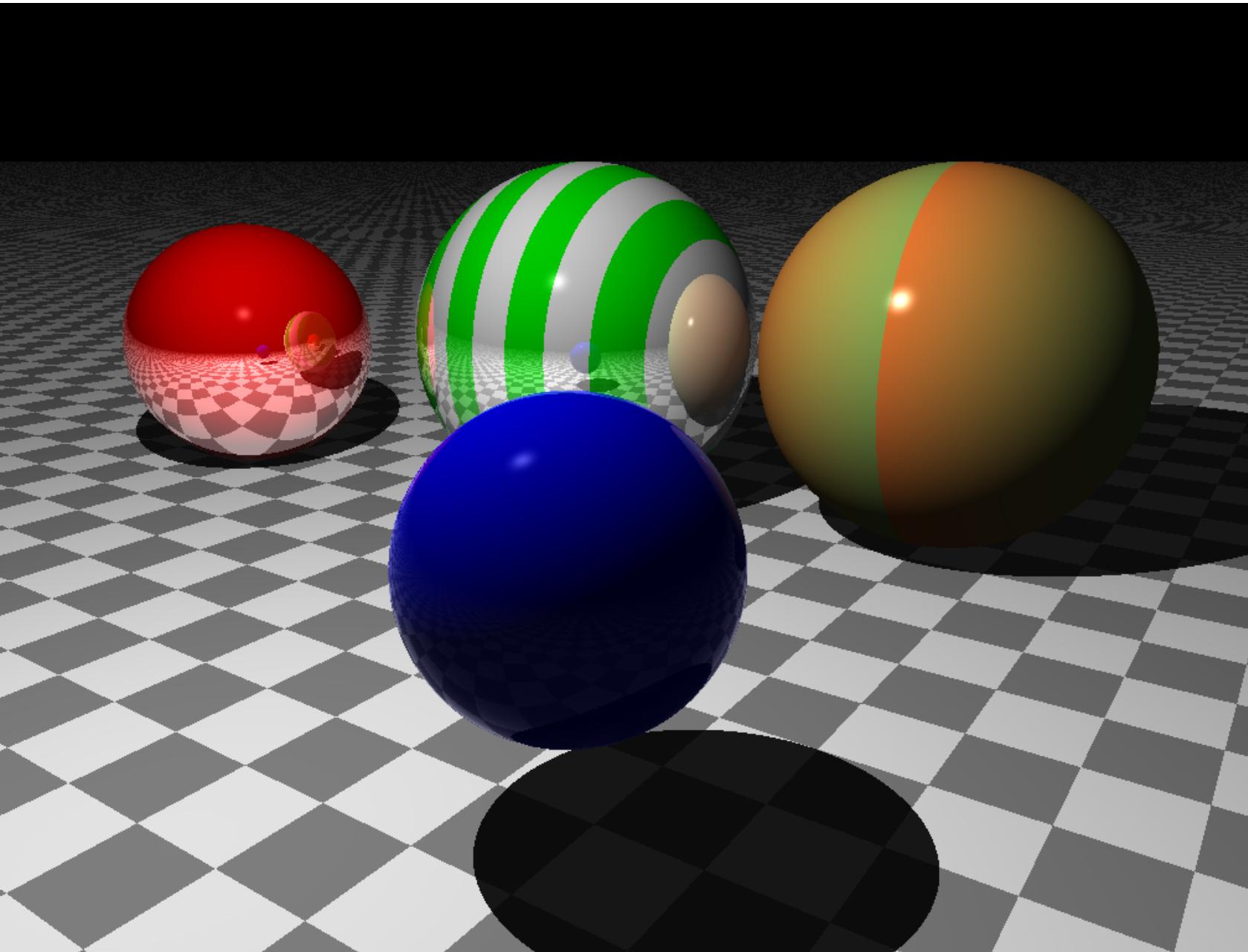
# AND IN MAIN

*Provide the layers*



```
val world: ZLayer[ATModule, Nothing, WorldModule] =  
  (topologyM ++ hitCompsM ++ phongM) >>> worldModule.live  
  
val rastering: ZLayer[ATModule, Nothing, RasteringModule] =  
  (world ++ cameraModule.live) >>> rasteringModule.chunkRasteringModule  
  
val full: ZLayer.NoDeps[Nothing, Rastering] = (layers.atM >> rastering)  
  
object Main extends zio.App {  
  
  override def run(args: List[String]): ZIO[ZEnv, Nothing, Int] =  
    httpProgram.provideLayer(full)  
}
```

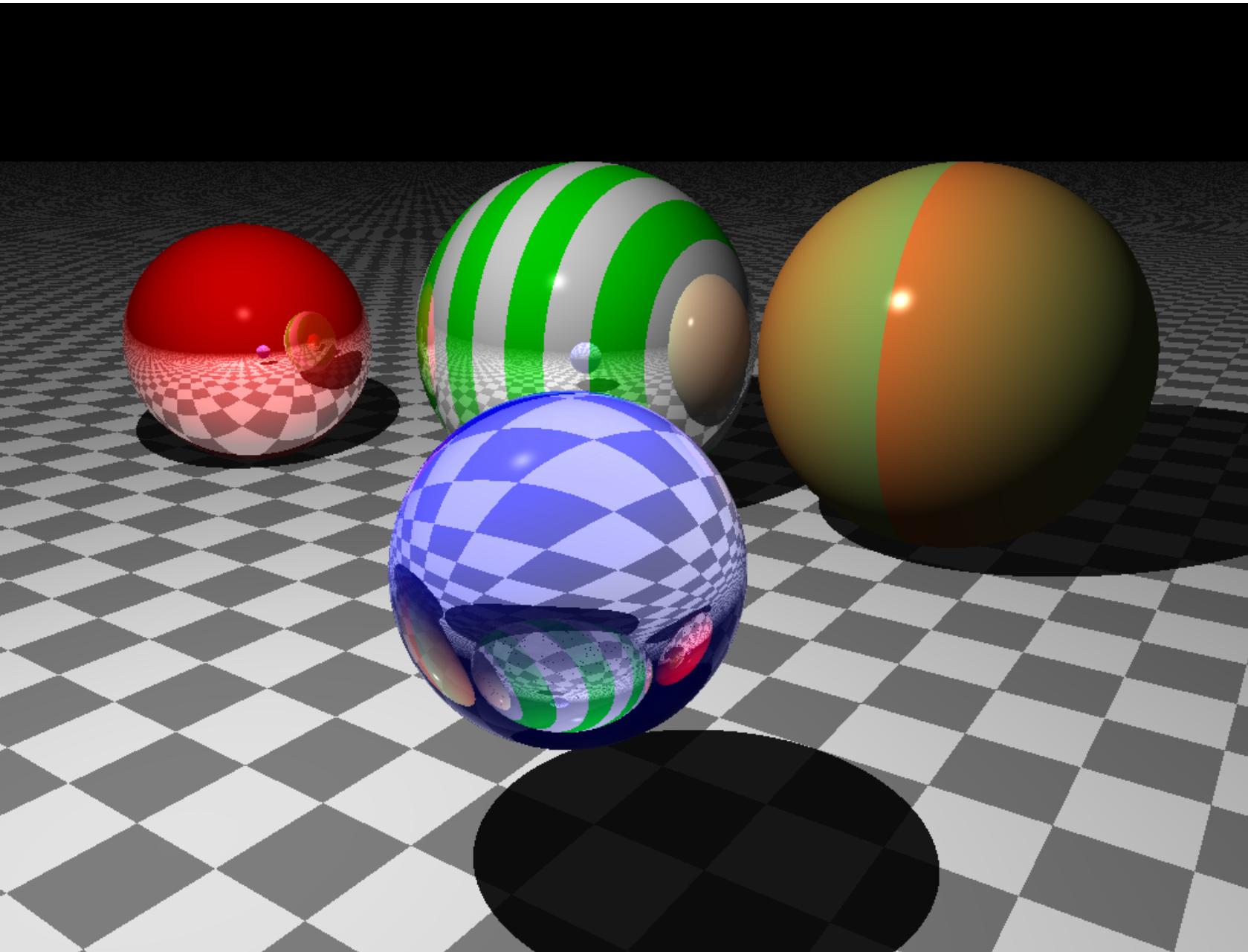
## *Swapping modules*



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

```
val world: ZLayer[ATModule, Nothing, WorldModule] =  
  (topologyM ++ hitCompsM ++ phongM) >>> worldModule.opaque
```

## *Swapping modules*



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

```
val world: ZLayer[ATModule, Nothing, WorldModule] =  
  (topologyM ++ hitCompsM ++ phongM) >>> worldModule.live
```

# CONCLUSION - *ZLayer*

# CONCLUSION - *ZLayer*

- ▶ Dependency graph in the code 💪

# CONCLUSION - ZLayer

- ▶ Dependency graph in the code 💪
- ▶ Type safety, no magic 🙌

# CONCLUSION - ZLayer

- ▶ Dependency graph in the code 💪
- ▶ Type safety, no magic 🙌
- ▶ Compiler helps to satisfy requirements 😊

# CONCLUSION - ZLayer

- ▶ Dependency graph in the code 💪
- ▶ Type safety, no magic 🙌
- ▶ Compiler helps to satisfy requirements 😊
- ▶ Try it out, and join ZIO Discord channel 😊

*Thank you!*

 @pierangelocecc

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