

# Path Tracer

Informática gráfica

Ingeniería informática

Jorge Solán Morote

Francisco Javier Pizarro

23/12/2023

# Índice

<b>Índice</b>	<b>2</b>
<b>Capítulo 1</b>	<b>3</b>
<b>Capítulo 2</b>	<b>4</b>
<b>Diseño del algoritmo</b>	<b>4</b>
2.1 Ecuación de render	4
<b>Capítulo 3</b>	<b>5</b>
<b>Implementación del algoritmo</b>	<b>5</b>
3.1 Ecuación de render	5
3.2 Implementación de las extensiones	6
3.2.1 Geometrías adicionales	6
3.2.1.1 Rectángulo	6
3.2.1.2 Triángulo	7
3.2.1.3 Mallas de triángulos - OBJ	7
3.2.2 BRDF de Phong	8
3.2.3 Carga de objetos 3D	9
3.2.4 Formatos de salida adicionales	10
3.2.5 Medios participativos	11
3.2.6 Texturas	13
3.2.7 Ecuaciones de Fresnel	15
3.2.8 Bump Mapping	16
3.2.9 Enfoque de cámara	17
3.2.10 Documentación profesional	18
3.3 Mejoras de rendimiento	19
3.3.1 Concurrencia y distribución	19
3.3.2 BVH	21
3.3.3 LOD	21
3.3.4 Lazy Eval / Strict Eval	22
3.3.5 Optimización a nivel de compilado	23
3.3.6 Pre Cálculo de estructuras	24
<b>Capítulo 4</b>	<b>25</b>
<b>Resultados experimentales</b>	<b>25</b>
<b>Capítulo 5</b>	<b>32</b>
<b>Conclusiones</b>	<b>32</b>
<b>Bibliografía</b>	<b>33</b>

# Capítulo 1

Este proyecto ha sido desarrollado en el marco de la asignatura de informática gráfica, este consiste en comprender las bases teóricas del funcionamiento de la generación de imágenes empleando para ello técnicas de Path Tracing para lo relativo a la simulación de la luz. Una vez comprendida la base teórica se han aplicado estos conocimientos programando nuestro propio generador de renders basado en Path Tracing.

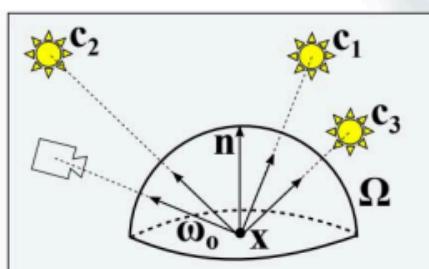
Cabe destacar la decisión de implementar dicho código en el lenguaje de programación [Haskell](#), de esta manera a parte de la evidente ventaja de legibilidad del código, nos aseguramos de evitar errores subyacentes causados por la manipulación inadecuada de variables, punteros o valores dentro del código, dado que Haskell es un lenguaje puramente funcional. Otra ventaja muy importante que nos ha brindado esta decisión a largo plazo es el emplear el denominado como CDD(Compiler Driven Development), es decir en cuenta de tener que buscar errores en tiempo de ejecución que pasan inadvertidos en tiempo de compilación, nos encontramos con que gracias a la inferencia de tipos estáticos del lenguaje y al paradigma funcional el compilador encuentra errores que en otros lenguajes pasarían desapercibidos a priori.

# Capítulo 2

## Diseño del algoritmo

### 2.1 Ecuación de render

La ecuación de render consiste en mirar en la semiesfera positiva de un punto toda la luz recibida de las fuentes de luz y otras figuras de la escena además de cómo esta afecta al propio objeto, sin embargo al ser una función con integral es imposible calcularlo en tiempo razonable, por ello se opta por transformar la ecuación de render como un sumatorio de la siguiente manera:


$$L_o(\mathbf{x}, \omega_o) = \sum_{i=1}^n \frac{p_i}{|\mathbf{c}_i - \mathbf{x}|^2} f_r \left( \mathbf{x}, \frac{\mathbf{c}_i - \mathbf{x}}{|\mathbf{c}_i - \mathbf{x}|}, \omega_o \right) \left| \mathbf{n} \cdot \frac{\mathbf{c}_i - \mathbf{x}}{|\mathbf{c}_i - \mathbf{x}|} \right|$$

Sin embargo esto solo hace referencia a la luz directa del punto en el que miramos, para calcular toda la luz (Luz directa + luz indirecta) se desarrollan técnicas como el Path Tracer.

El Path Tracer realiza lo siguiente, una vez el rayo de la cámara choca con un material decide su próxima interacción dependiendo de las propiedades del material, ya sea difuso, especular o refractante, en una máquina con mucho poder computacional podrían hacerse los 3 casos a la vez pero el número de rebotes a calcular crecería exponencialmente, por ello se realiza la “Ruleta Rusa” donde se elige por probabilidad su siguiente comportamiento.

Si el resultado es especular, se lanza el rayo con dirección especular en el objeto golpeado.

Si sale refractante lo mismo que con el anterior.

Si embargo si el resultado es difuso se lanza el siguiente rayo de manera aleatoria simulando el camino que podría llegar a recorrer un fotón. Cuando más muestras por caminos se tomen, más exacto llegará a ser el render en comparación con la primera integral descrita.

# Capítulo 3

## Implementación del algoritmo

### 3.1 Ecuación de render

Para realizar el render se ha optado por, con solo luces de área se realiza muestreo del coseno para lanzar el siguiente rayo, en cambio en luces puntuales se opta por muestreo del ángulo sólido.

Como todo el programa es recursivo, con luces puntuales calculamos una vez al principio la luz directa para toda la escena, y después se lanza la luz indirecta tantas veces como el número de ppp simulando sus diferentes caminos.

```
listRayToRGB :: [Luz] -> Set.Set Shape -> [Ray] -> StdGen -> Int -> [RGB]
listRayToRGB luz figuras rayos gen nRay = zipWith (+) colorDirecto $ map (`divRGB` fromIntegral ppp) colorIndirecto

where
    antial = map listRay $ parametricShapeCollision figuras rayos
    rayColisions = antialiasing nRay antial

    (gens, _) = splitAt (length rayColisions * ppp) $ drop 1 $ iterate (snd . split) gen -- Semillas

    ppp = 256 -- Caminos por pixel
    colorIndirecto = zipWith (pathTracer 1 luz figuras ppp) rayColisions gens
    colorDirecto = map (luzDirecta luz figuras) rayColisions
```

El código de luz directa es simplemente mirar los puntos de luz y calcular la luz con la ecuación de render del apartado anterior con la potencia de la luz, la brdf y el término del coseno. En cambio el código de luz indirecta es el que tiene más implicaciones ya que simulará el camino (Path) del fotón en la escena. En él lo primero que se hace es mirar la Ruleta Rusa que dicta la interacción que se debe simular, difuso, especular, refractante o absorción, especular y refractante simplemente mira el siguiente objeto con el que chocan y repiten la luz indirecta, si es difuso se busca un el siguiente objeto con el muestreo uniforme del ángulo sólido, así que se tiene que multiplicar por su probabilidad, y después solo se comprueba cómo le afecta el siguiente objeto al actual.

Además de todo esto, para conocer la luz que tiene el objeto actual realizamos next-event-estimation con la llamada a luzDirecta en los casos correspondientes.

```

luzIndirecta :: Obj -> [Luz] -> Set.Set Shape -> StdGen -> RGB
luzIndirecta obj luz figuras gen = result where
  result = case caso of
    0 -> rgbNew rndObj (brdf obj figuras) `modRGB` por
    1 -> colorIndirecto objCr `modRGB` por
    2 -> colorIndirecto objEsp `modRGB` por
    _ -> RGB 0 0 0

  (caso, por) = ruletaRusa (trObj obj) gen

  colorDirecto nxtObj = luzDirecta luz figuras nxtObj `modRGB` abs ((w0Obj nxtObj .*
normObj obj) * 2 * pi)

  colorIndirecto nxtObj = luzIndirecta nxtObj luz figuras gen'

  rgbNew nxtObj = formula (colorDirecto nxtObj + colorIndirecto nxtObj) 1 (colObj obj)
  (colObj nxtObj) (normObj nxtObj)

  figuras' = Set.filter (\shape -> idObj obj /= getShapeID shape) figuras
  rndObj = objAleatorio figuras' obj gen

  gen' = snd (split gen)

  (objCr, rFlNew) = objCristal figuras' (w0Obj obj) (normObj obj) 1 (reflObj obj) (colObj
obj)
  objEsp = objEspejo figuras' (w0Obj obj) (normObj obj) (colObj obj)

```

Con luz de área se realiza exactamente lo mismo como si toda la escena fuera con luz indirecta, no se calcula su luz directa.

El código es muy parecido al anterior simplemente que aquí se mira si el objeto con el que se choca es una luz de área, y si es así se le aporta la luz de este objeto, no se realiza next-event-estimation ya que la luz solo se transmite de esta manera, y por usar en este caso el muestreo del coseno, en vez de tener que multiplicar el camino del objeto difuso por  $2\pi\cos$  se le multiplica por  $\pi$  por ser su probabilidad

```

luzAreaRec :: Set.Set Shape -> Obj -> StdGen -> RGB
luzAreaRec figuritas obj gen = rgbFin
where
  rgbFin = if idObj obj == 4 then RGB 1 1 1 else if rgbObj rndObj == RGB 0 0 0 then RGB 0
0 0 else if nanRGB result then RGB 0 0 0 else result
  result = case caso of
    0 -> rgbNew rndObj (brdf obj figuritas) `modRGB` (pi * por)
    1 -> luzAreaRec figuritas objCr gen' `modRGB` por
    2 -> luzAreaRec figuritas objEsp gen' `modRGB` por
    _ -> RGB 0 0 0

(caso, por) = ruletaRusa (trObj obj) gen

  rgbNew nxtObj = formula (luzAreaRec figuritas nxtObj gen') 1 (colObj obj) (colObj nxtObj)
  (normObj nxtObj)

  figuritas' = Set.filter (\shape -> idObj obj /= getShapeID shape) figuritas
  rndObj = objAleatorio figuritas' obj gen

  gen' = snd (split gen)

  (objCr, rFlNew) = objCristal figuritas' (w0Obj obj) (normObj obj) 1 (reflObj obj) (colObj
obj)
  objEsp = objEspejo figuritas' (w0Obj obj) (normObj obj) (colObj obj)

```

## 3.2 Implementación de las extensiones

### 3.2.1 Geometrías adicionales

Una primera forma de aportar una mayor riqueza visual a las imágenes generadas es mediante el uso de otras geometrías base. La más importante es el triángulo dado que esta permite construir otras más complejas usándola como base, no obstante se han implementado también las siguientes primitivas:

#### 3.2.1.1 Rectángulo

En primer lugar se ha desarrollado el rectángulo o plano finito, principalmente para generar imágenes con luces de área o para objetos texturizados como cuadros, o paredes.

Para la especificación del rectángulo este se define como el punto central, su altura, su anchura, su vector normal y su vector tangente que indica hacia donde está orientado.

El código que realiza la colisión del rectángulo con un rayo es el siguiente:

```
oneCollision (Rectangle (Rectangulo {..})) (Ray {..})
| denom /= 0 && t > 0 && withinBounds = (Obj t rgbRe dR (dirPoint collisionPoint) normRe'
trRe reflRe idRe)
| otherwise = (Obj (-1) (RGB 0 0 0) dR (Point3D 0 0 0) (Direction 0 0 0) trRe reflRe 0)
where
  offset = collisionPoint - pointDir centRe
  localX = offset .* right
  localY = offset .* up
  halfWidth = ancRe / 2
  halfHeight = altRe / 2
  collisionPoint = pointDir oR + escalateDir t dR
  t = (centRe #< oR) .* normRe / denom
  withinBounds = -halfWidth <= localX && localX <= halfWidth && -halfHeight <= localY &&
localY <= halfHeight
  denom = dR .* normRe
  right = normal tngRe
  up = normRe * right
  normRe' = if dR .* normRe > 0 then normal (escalateDir (-1) normRe) else normal normRe
```

### 3.2.1.2 Triángulo

Se ha desarrollado también la geometría del triángulo, principalmente por su facilidad de uso en mallas de triángulos y archivos obj.

Sobre su intersección se ha optado por desarrollarlo con el algoritmo de [Möller–Trumbore](#).

El siguiente código implementa la colisión de un rayo con un triángulo:

```
oneCollision tr@(Triangle (Triangulo {})) (Ray {}) =  
    case ray1TriangleIntersection oR dR p0Tr p1Tr p2Tr of  
        Just (t, intersectionPoint) ->  
            let normalVec = (p1Tr #< p0Tr) * (p2Tr #< p0Tr)  
                normalVec' = if (dR.*normalVec) > 0 then normal (escalateDir (-1) normalVec) else  
normal normalVec  
            in (Obj t rgbTr dR intersectionPoint normalVec' trTr reflTr idTr tr)  
        Nothing -> (Obj (-1) (RGB 0 0 0) dR (Point3D 0 0 0) (Direction 0 0 0) (0,0,0) 0 0 tr)
```

### 3.2.1.3 Mallas de triángulos - OBJ

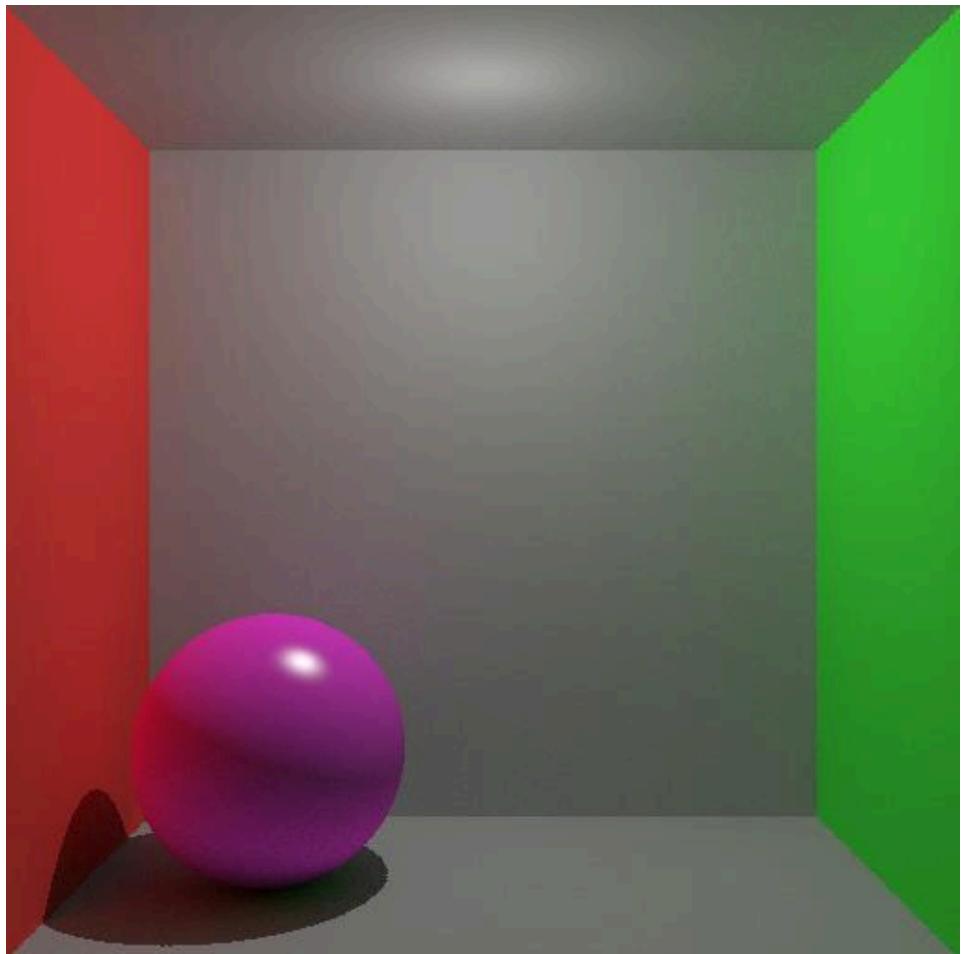
Dado que un obj contiene una gran cantidad de triángulos que conforman un mismo objeto, se ha decidido tratar estos como tal, para ello se define la geometría de la malla de triángulos (esta definición va estrechamente ligada a la optimización asociada a la mism BVH). Para la carga de esta geometría se ha optado por el formato de archivo .obj donde se definen los vértices de los triángulos, las distintas figuras asociadas y las coordenadas (u,v) asociadas a cada figura, esto último es necesario para poner texturas a las mallas de triángulos.

No se adjunta aquí el código de esta geometría, dado que este va ligado a la mejora de rendimiento asociado al mismo.



### 3.2.2 BRDF de Phong

La BRDF de Phong trabaja sobre un material difuso aportándole una apariencia mate, esta se la dá tirando un rayo desde la luz al objeto y se obtiene la dirección resultante como si el objeto fuera un espejo (dirección especular). Con esta dirección y el ángulo de incidencia en el objeto obtenemos la primera parte de la fórmula, la segunda es con un alpha (propiedades del material) dado que indicará sobre todo cuánta potencia lumínica tiene el brillo nuevo formado y el radio que se forma. En la siguiente imagen se aprecia el efecto obtenido:



Código que implementa la BRDF de Phong:

```
fPhong :: Point3D -> Obj -> Float -> Set.Set Shape -> Float
fPhong pLuz obj alpha figuras = if col then (alpha+2/2) * abs (dirEspejo (colObj obj #<pLuz)
(normObj obj) .* w0Obj obj)**alpha else 0
where
  col = colision (colObj obj) pLuz figuras'
  figuras' = Set.filter (\shape -> idObj obj /= getShapeID shape) figuras
```

### 3.2.3 Carga de objetos 3D

Un punto fundamental para poder crear un buen render es la posibilidad de emplear objetos 3D creados por otras herramientas, para lograr introducir dichos objetos en el render es necesario programar una serie de funciones para poder leer y parsear los distintos ficheros que contienen la información relativa a estos objetos 3D, por simplicidad y popularidad se ha elegido como formato de carga el **OBJ**.

Un OBJ a grandes rasgos está conformado por información de dos elementos, los vértices los cuales tienen unas coordenadas X Y Z, y las caras las cuales están formadas por 3 o 4 vértices, si bien existen algunos aspectos adicionales dentro de este formato como pueden ser las normales, por evitar complicaciones innecesarias se ha optado por implementar la lectura del formato más simple posible, adicionalmente también cuenta con unas coordenadas U V para representar las texturas.

Dado que al generar un OBJ desde otras herramientas como Blender, este puede ser algo más complejo de lo esperado, se ha creado un script auxiliar en Python el cual simplifica dichos OBJs al formato esperado.

De nuevo por buenas prácticas y modularidad se ha aislado el código encargado de la lectura e interpretación de ficheros de esta índole. A continuación se muestra el código encargado de interpretar el contenido de los OBJs:

```
loadObjFile :: FilePath -> IO ([Point3D], [TrianglePos])
loadObjFile filePath = do
    contents <- readFile filePath
    let lines' = lines contents
        (vertices, triangles) = foldr splitLines ([], []) lines'
        validVertices = mapMaybe parsePoint3D lines'
        validTriangles = mapMaybe parseTriangle lines'
    return (validVertices, validTriangles)
where
    splitLines line (vertices, triangles)
        | null (words line) = (vertices, triangles)
        | Just vertex <- parsePoint3D line = (vertex : vertices, triangles)
        | Just triangle <- parseTriangle line = (vertices, triangle : triangles)
        | otherwise = (vertices, triangles)
```

Dado que es interesante modificar un objeto una vez ha sido cargado, también se adjunta un ejemplo de la manipulación de este tras ser extraído del fichero .obj:

```
let objFilePath2 = "../meshes/simplehaskell.obj"
(vertices2, trianglesH2) <- loadObjFile objFilePath2
let vertices2' = map (escalatePointt (1).movePoint (Direction (-5) (-5) (-28)).rotatePointt 'Y' (90)) vertices2
customTrianglesH2 = convertToCustomFormat (RGB 122 10 255) (0.85, 0,0) 0 (vertices2', trianglesH2)
boundingVol'' = buildBVH 4000 customTrianglesH2
figuras''' = Set.fromList $ addFigMult [(Accelerator boundingVol'')] (Set.toList figuras'')
```

### 3.2.4 Formatos de salida adicionales

Existen muchos formatos posibles de salida para las imágenes generadas, uno de ellos es el formato .ppm, el cual es el requerido por la asignatura, adicionalmente se ha decidido implementar como formato de salida extra el formato .bmp, dada su simplicidad.

Aquí se adjunta el código responsable de guardar las imágenes generadas en dicho formato:

```
writeBMP :: FilePath -> Int -> Int -> BS.ByteString -> IO ()  
writeBMP filename width height customPixelData = do  
    let fileSize = 54 + 4 * width * height  
    let pixelDataOffset = 54  
    let dibHeaderSize = 40  
    let bitsPerPixel = 32  
    let compressionMethod = 0  
    let imageSize = 4 * width * height  
    let xPixelsPerMeter = 2835 -- 72 DPI  
    let yPixelsPerMeter = 2835 -- 72 DPI  
  
    BS.writeFile filename $ BS.concat  
        [ BS.pack [66, 77] -- Signature ("BM")  
        , intTo4Bytes fileSize -- File size  
        , intTo4Bytes 0 -- Reserved  
        , intTo4Bytes pixelDataOffset -- Pixel data offset  
        , intTo4Bytes dibHeaderSize -- DIB header size  
        , intTo4Bytes width -- Image width  
        , intTo4Bytes height -- Image height  
        , intTo2Bytes 1 -- Number of color planes  
        , intTo2Bytes bitsPerPixel -- Bits per pixel  
        , intTo4Bytes compressionMethod -- Compression method  
        , intTo4Bytes imageSize -- Image size  
        , intTo4Bytes xPixelsPerMeter -- Horizontal resolution (pixels per meter)  
        , intTo4Bytes yPixelsPerMeter -- Vertical resolution (pixels per meter)  
        , BS.replicate 8 0 -- Reserved  
        , customPixelData -- White pixel data  
        ]
```

### 3.2.5 Medios participativos

La implementación de la niebla homogénea se ha realizado simulando un efecto de postprocesado sobre la imagen final sobre el render.

Su funcionamiento consta de dos partes, en la primera se calcula la pérdida de visibilidad del rayo a través de la niebla de la siguiente manera:

## RTE – *Extinction on a finite beam*

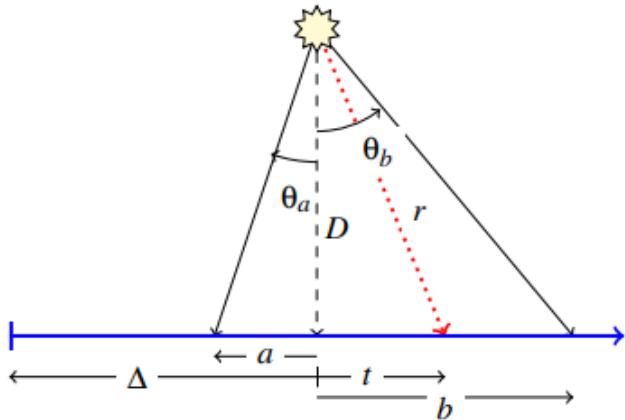
$$\begin{aligned}
 dL(\mathbf{x}, \omega) &= -\mu_t(\mathbf{x}, \omega)L(\mathbf{x}, \omega) dz && \text{>> Assuming extinction only} \\
 \frac{dL(\mathbf{x}, \omega)}{L(\mathbf{x}, \omega)} &= -\mu_t(\mathbf{x}, \omega) dz && \text{>> Reorganizing } dz \\
 \ln(L(\mathbf{x}_z, \omega)) - \ln(L(\mathbf{x}_0, \omega)) &= - \int_{\mathbf{x}_0}^{\mathbf{x}_z} \mu_t(\mathbf{x}, \omega) dz && \text{>> Integrating from } \mathbf{x}_0 \text{ to } \mathbf{x}_z \\
 \ln\left(\frac{L(\mathbf{x}_z, \omega)}{L(\mathbf{x}_0, \omega)}\right) &= - \int_{\mathbf{x}_0}^{\mathbf{x}_z} \mu_t(\mathbf{x}, \omega) dz && \text{>> ...or equivalently} \\
 T(\mathbf{x}_0 \leftrightarrow \mathbf{x}_z) &= \frac{L(\mathbf{x}_z, \omega)}{L(\mathbf{x}_0, \omega)} = e^{- \int_{\mathbf{x}_0}^{\mathbf{x}_z} \mu_t(\mathbf{x}, \omega) dz} && \text{>> Apply the exponential...}
 \end{aligned}$$

Como simplemente está implementada la niebla homogénea la derivada anterior se reduce a una constante, quedando la extinción de la luz sobre el rayo finito así:

– In **homogeneous** media:

$$\begin{aligned}
 T(\mathbf{x}_0 \leftrightarrow \mathbf{x}_z) &= e^{- \int_{\mathbf{x}_0}^{\mathbf{x}_z} \mu_t dz} && \text{>> Constant extinction} \\
 &= e^{-\mu_t z}
 \end{aligned}$$

Para la aportación de la luz en el rayo por parte de la niebla se ha optado por computarlo con [Equiangular Sampling](#). En el anterior paper se explica, que, en vez de recoger las muestras de los puntos en el rayo de visión con la Montecarlo como se hace en “Mean Free Path Sampling”, las muestras que se recogen con Equiangular Sampling están más concentradas en aquellos puntos donde llega la luz en vez de ser tan aleatorio



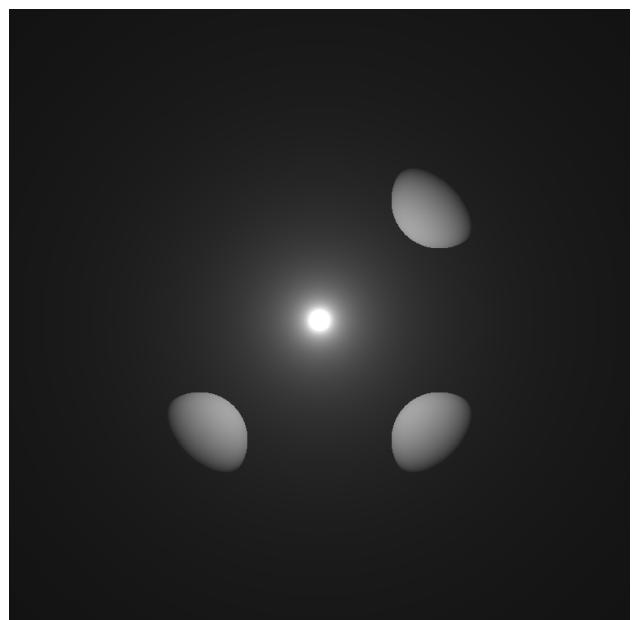
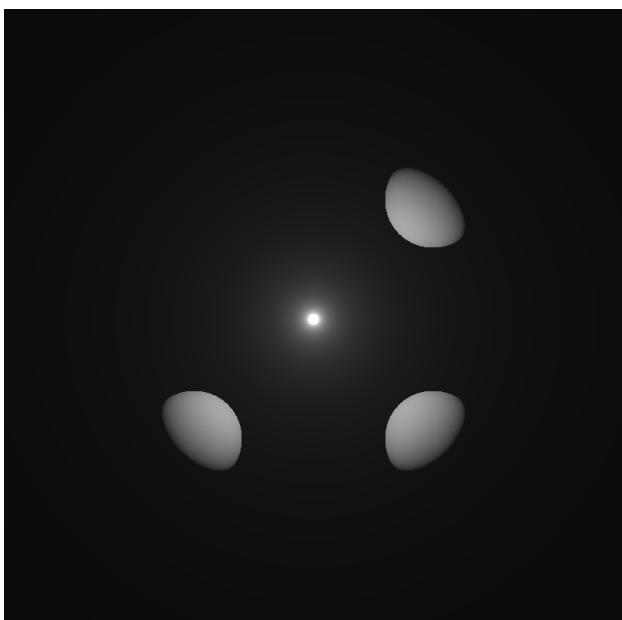
Sin embargo, como la luz renderizada es total se tuvo que tomar una aproximación distinta, esta consiste en que, para añadirle el efecto de niebla, en el rayo trazado al objeto, se halla el punto más cercano a la luz, se calcula la distancia a esta, y se eleva a -2.

```

addNiebla :: Luz -> Obj -> Float -> Set.Set Shape -> RGB -> RGB
addNiebla (Luz {..}) obj x figuras rgb = if (mindObj obj) < 0 then RGB 0 0 0 else newRGB +
(rgb `modRGB` reducObj)
where
    rgb' = head $ gammaFunc' 1 2.4 [rgb]
    newRGB = RGB fact fact fact * (scale luzRGB)
    reducLuz = if zP closest < 0 then exp (x * zP closest / 30) else 1
    reducObj = if zP (colObj obj) < 0 then exp ((1-x) * zP (colObj obj) / 30) else 1 -- Como
le afecta la niebla de lejos a los objetos

    camP = Point3D 0 0 35 -- Comienzo de la camara
    cam = Ray camP dir
    dir = normal $ (colObj obj) #< camP
    closest = distanceToRay luzP cam
    fact = (1-x) * reducLuz * (distPoint luzP closest ** (-2)) -- Como afecta la luz a los
objetos
  
```

Las siguientes imágenes son ambas tomadas con niebla homogénea pero modificando la densidad de “partículas” que se encuentran dentro de esta niebla.



### 3.2.6 Texturas

Con el objetivo de poder generar unas imágenes más diversas, se ha decidido añadir la posibilidad de cargar texturas para algunas de las geometrías ( Esferas, triángulos y rectángulos).

La idea general es, que para una colisión en el objeto y el objeto al que hace referencia, se calculan los valores u v que posteriormente se utilizarán para en la textura original para saber a qué píxeles se refiere de una imagen png que se le asocia. A continuación se adjuntan los fragmentos de código asociados a dicho cálculo.

```
getUV (Rectangle(Rectangulo {...})) p = (u,v)
where
    halfHeight = altRe / 2
    halfWidth = ancRe / 2
    Direction x y z = normRe
    right = tngRe
    up = normal $ normRe * normal right
    bottomLeft = calculateVertex (-halfWidth) (-halfHeight)
    bottomRight = calculateVertex halfWidth (-halfHeight)
    topLeft = calculateVertex (-halfWidth) halfHeight

    calculateVertex w h = centRe `addPoints` dirPoint (escalateDir w right + escalateDir h
up)

    !u = distanceToRay p (Ray bottomLeft (bottomLeft #< topLeft)) / altRe
    !v = distanceToRay p (Ray bottomLeft (bottomLeft #< bottomRight)) / ancRe
```

El rectángulo al definirlo de la manera que se hace, si se le quiere meter una textura se necesitan hallar sus direcciones desde el punto de más abajo izquierda a sus dos puntos más cercanos del rectángulo y, depende de la altura y anchura del triángulo, con esas cosas se pueden calcular sus puntos u,v.

```
getUV (Sphere (Esfera {...})) p = (u,v)
where
    (Point3D x y z) = p
    (Point3D cx cy cz) = centEs
    u = 0.5 + atan2 (z - cz) (x - cx) / (2 * pi)
    v = 0.5 - asin ((y - cy) / radEs) / pi
```

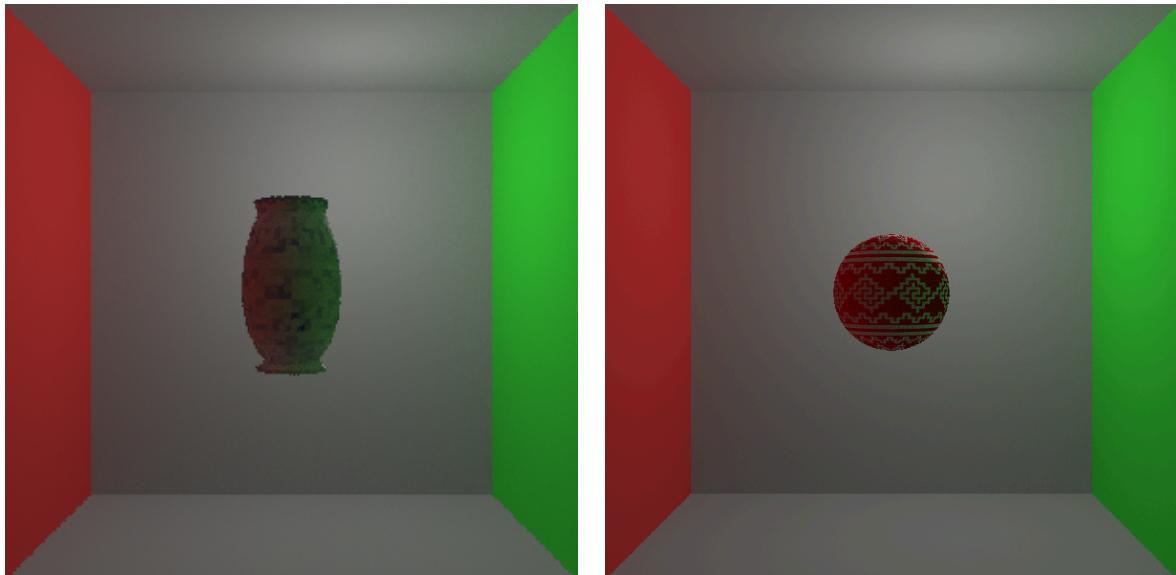
Para la esfera es necesario conocer los puntos polares (latitud y longitud) en los que se ha colisionado el punto, con esto, ya se puede hallar el u y el v.

```
getUV (Triangle (Triangulo {...})) p = (u,v)
where
    v0 = p2Tr #< p0Tr
    v1 = p1Tr #< p0Tr
    v2 = p #< p0Tr
    n = v1 * v0
    cross1 = v2 * v1
    cross2 = v0 * v2
    aTri = 0.5 * modd n
    alpha = modd cross1 / (2 * aTri)
    beta = modd cross2 / (2 * aTri)
    gamma = 1 - alpha - beta
    u = (alpha * uP uv0Tr + beta * uP uv1Tr + gamma * uP uv2Tr)
    v = (alpha * vP uv0Tr + beta * vP uv1Tr + gamma * vP uv2Tr)
```

Finalmente para el triángulo se halla el u y el v gracias a sus coordenadas baricéntricas que nos dicen cuánto se aleja cada punto del centro del triángulo.

Para las mallas de triángulos se termina usando el uv del triángulo, sin embargo en la carga de este se especifica también el triángulo que hace referencia en su textura, que en el código vendría dado por los puntos uv0Tr, uv1Tr y uv2Tr, así que no tiene complicación alguna más que realizar bien la carga de estos.

A continuación se dejan algunas pruebas realizadas con texturas.



### 3.2.7 Ecuaciones de Fresnel

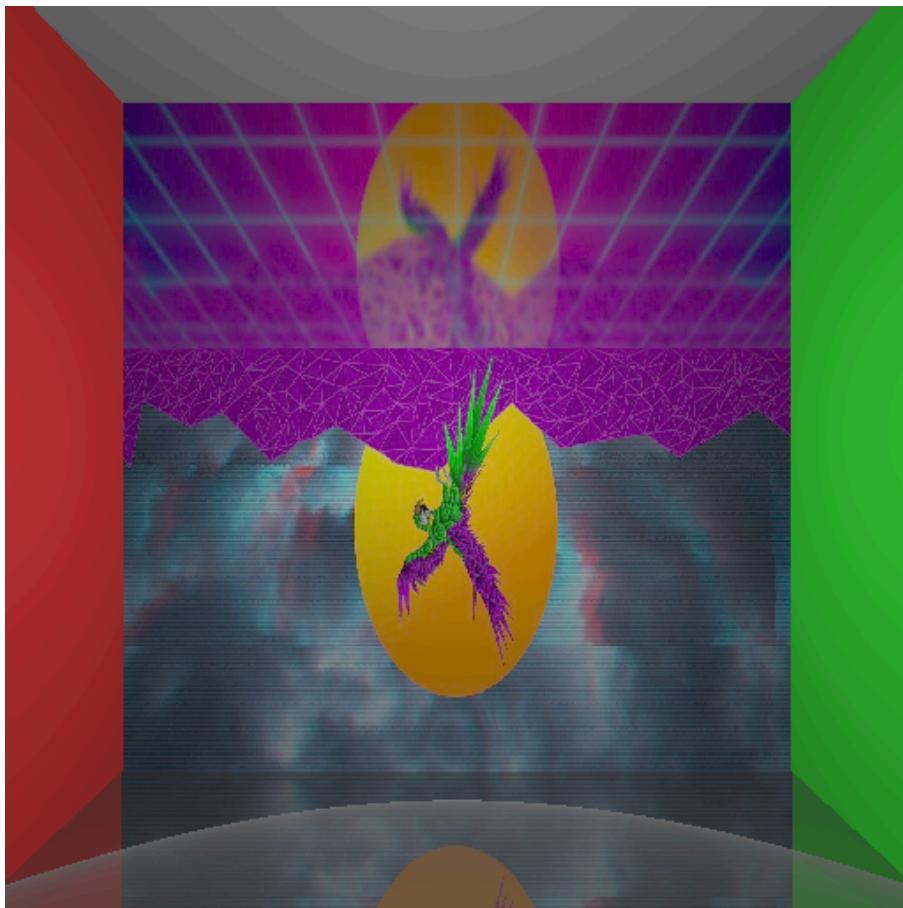
Para sacar la siguiente imagen se han modificado las propiedades de la cámara para poder medir bien su efecto sobre un suelo con [ecuaciones de fresnel](#) implementadas para que se vea el efecto, las ecuaciones son las siguientes:

Parallel Pol. Component

$$\rho_{\parallel} = \frac{\eta_1 \cos \theta_i - \eta_0 \cos \theta_t}{\eta_1 \cos \theta_i + \eta_0 \cos \theta_t} \quad \text{reflected: } F_r = \frac{1}{2} (\rho_{\parallel}^2 + \rho_{\perp}^2)$$

Perpendicular Pol. Component

$$\rho_{\perp} = \frac{\eta_0 \cos \theta_i - \eta_1 \cos \theta_t}{\eta_0 \cos \theta_i + \eta_1 \cos \theta_t} \quad \text{refracted: } F_t = 1 - F_r$$



Al observar la imagen podemos ver cómo, cuanto más al borde final del suelo estamos, se reduce su componente difuso (en verdad es refractante pero para que se viera en la prueba se hizo con difuso) y cuanto más alejado más se muestra, el efecto que se quiere conseguir con esto es cuando se mira hacia un cristal, cuanto más paralela tienes la visión con su tangente más se asemeja al comportamiento de un espejo, en cambio si lo miras directamente actúa casi por completo como un cristal, aunque podrás ver también tu reflejo aunque con menor intensidad.

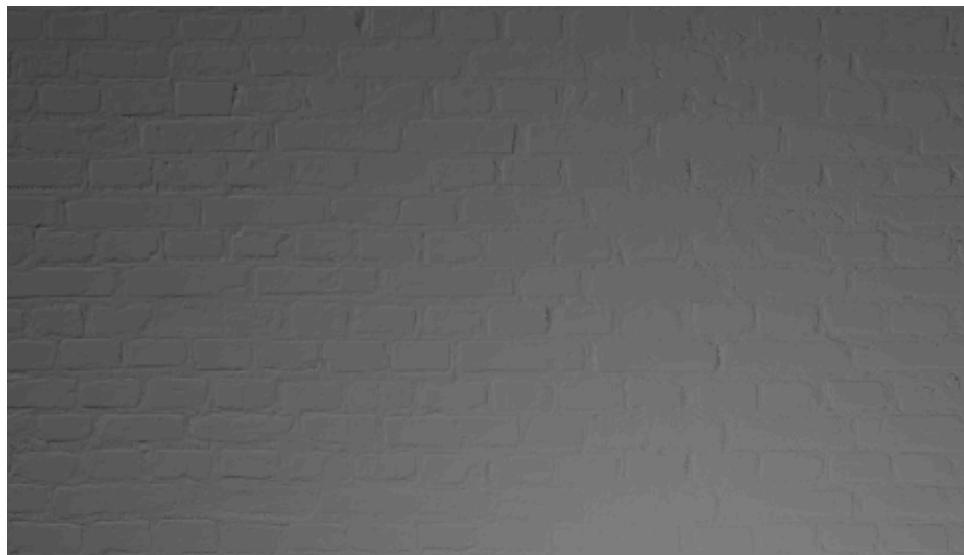
### 3.2.8 Bump Mapping

Para la realización del bump mapping primero se necesita, al menos en la versión implementada, una imagen en blanco y negro. Su implementación es similar a la de las texturas, primero se obtiene el pixel de la imagen del bump mapping equivalente en la figura y con [diferencias finitas](#), se realiza el cálculo de la nueva normal tal y como se indica en el [siguiente enlace](#).

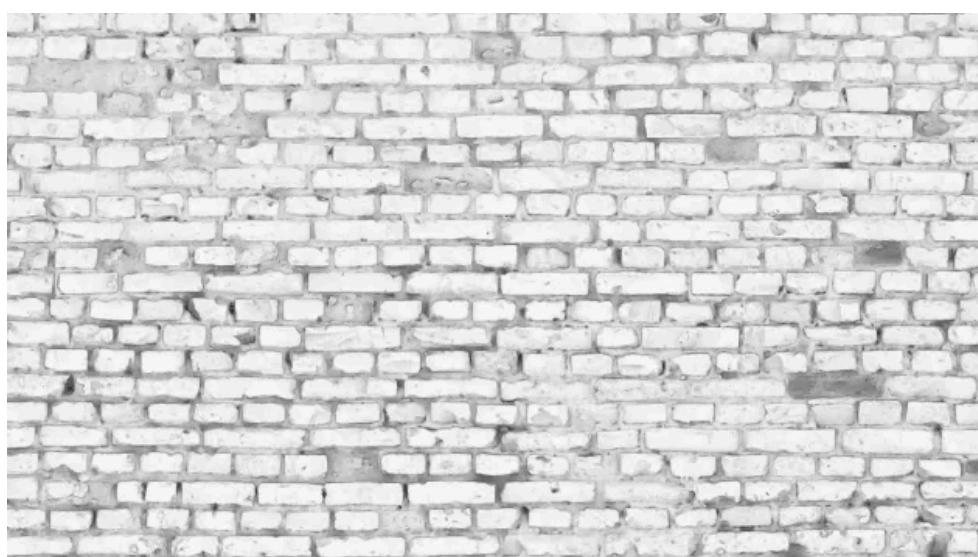
En resumidas cuentas se indica que, la nueva dirección de la normal, será vista desde un cambio de base sobre la normal de la figura, la Dirección( $B_u, 1, B_v$ ) normalizada. Los parámetros  $B_u$  y  $B_v$  se extraen de esas diferencias finitas, donde  $B_u$  es el valor del pixel, pasado de RGB a Float, de donde le corresponde el pixel a la figura - el pixel de la derecha. Con el parámetro  $B_v$  es exactamente lo mismo, pero con el pixel de arriba.

El resultado final es el siguiente:

Render resultante:

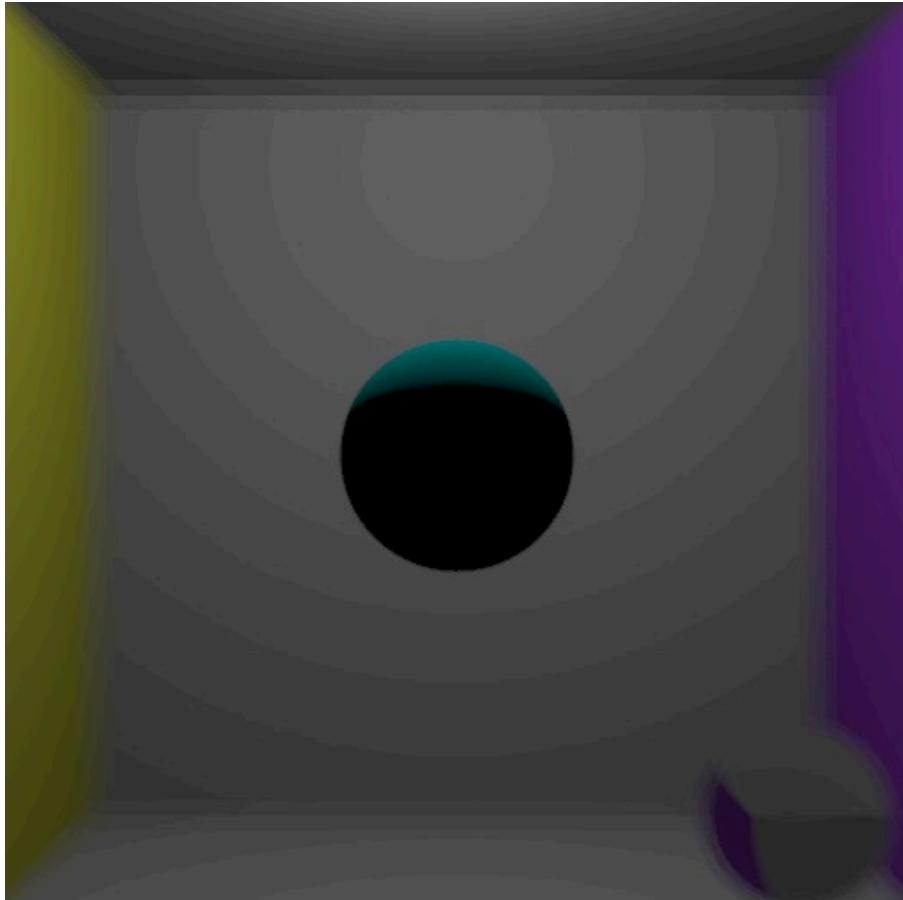


Textura de referencia:



### 3.2.9 Enfoque de cámara

Otra de los opcionales implementados es la posibilidad de enfocar un objeto y que el resto se vea algo difuminado. La idea es la siguiente, en vez de lanzar los rayos desde un solo punto se lanzan desde diferentes puntos como si fuera la apertura de una cámara y una vez sacados los distintos puntos, a la hora de mirar hacia qué dirección se lanzan los rayos se fija una dirección central que se utilizará como referencia para que todos, a una distancia ‘z’ apunten al mismo punto, pero antes de ese ‘z’ y después los rayos se separan y apuntan a otros puntos de la imagen, consiguiendo el siguiente efecto.



```
mulCam :: Camara -> Int -> Float -> [Camara]
mulCam cam@(Camara p b) n radio = cam : map (`Camara` b) (take n transformedPoints)
  where
    circlePoints = pointsInUnitCircle -- Tomamos todos los puntos dentro de un círculo
unitario

    -- Función para escalar y desplazar puntos según el círculo deseado
    transformPoint (x, y) = Point3D (radio * x) (radio * y) 20

    -- Lista de puntos dentro de un círculo unitario
    pointsInUnitCircle = [(cos theta, sin theta) | theta <- [0, (2 * pi) / fromIntegral n .. 2 * pi]]

    -- Aplicar la transformación a cada punto
    transformedPoints = map transformPoint circlePoints
```

### 3.2.10 Documentación profesional

Este aspecto adicional es algo muy distinto a los anteriores, para facilitar la comprensión del código de aquellos que por curiosidad quieren verlo se ha optado por documentar este de forma efectiva para generar una documentación profesional de Haskell accesible en formato web. Actualmente se encuentra desplegada en <https://render-haskell.duckdns.org/>.

La documentación permite de forma extremadamente sencilla consultar el funcionamiento concreto de una función o ver el propio código fuente. Este extra se escapa completamente del enfoque de la asignatura, no obstante se ha considerado que este aportaría, dada la peculiaridad del lenguaje, una mayor riqueza al proyecto para los curiosos.

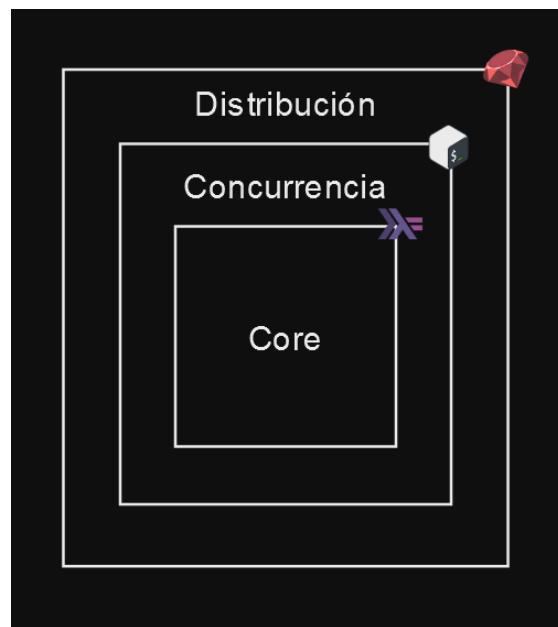
### 3.3 Mejoras de rendimiento

#### 3.3.1 Concurrency y distribución

La idea subyacente de esta mejora es bastante simple, se trata de emplear varios cores de una misma máquina y una vez logrado esto se trata de agregar un conjunto de cores empleando para ello varias máquinas.

Dada la naturaleza de Haskell no merecía la pena tratar de implementar desde el propio código del photon mapper la parte de distribución, no obstante si se realizó un estudio en profundidad para valorar si era técnicamente posible y viable implementar en este la concurrencia, a pesar de ser posible no era viable. Más concretamente Haskell cuenta con 2 bibliotecas estándar para cumplir este propósito la librería [Concurrency](#) y la librería [Parallel](#), la primera de ellas nos ofrece formas diversas de lanzar subprocessos no obstante no podemos obtener los resultados de estos de forma “transparente”, por el contrario la segunda librería **si** permite el lanzamiento de los denominados **sparks**(subprocesos ejecutados dentro del propio runtime de Haskell) los cuales **si** pueden compartir información de forma muy sencilla, a pesar de que la segunda alternativa parecía viable una vez realizamos las correspondientes pruebas pudimos comprobar que esa facilidad de comunicación tenía un coste alto en RAM, que escalaba significativamente conforme lo hacía el número de sparks por lo que esta solución tampoco era una opción adecuada.

Dado que las herramientas del propio lenguaje no cubrían nuestras necesidades optamos por abstraer el problema, es decir pasar el problema de la concurrencia a una capa superior y el problema de la distribución a otra sobre esta.

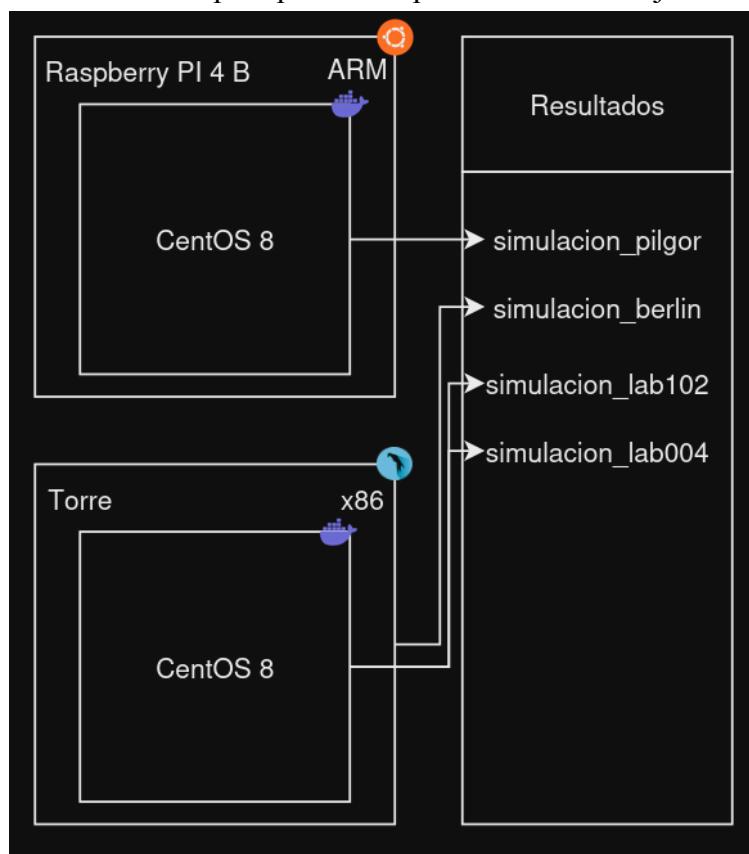


De esta forma primero tenemos el core programado en Haskell el cual acepta parámetros de ejecución, en la capa intermedia nos encontramos con un script de Bash el cual gestiona la concurrencia(emplea para ello procesos asíncronos y barreras), este script es el encargado de

inyectar las variables en la llamada al ejecutable principal. Por último en la capa exterior nos encontramos con un script de Ruby el cual contiene la información relativa a todas las máquinas a emplear(hostname,ip,cpus,nombre\_ejecutable), este va iterando por los Host definidos lanzando una serie de comandos por vía SSH que culminan con la ejecución del script bash de la capa subyacente inyectando a este las variables pertinentes.

Para que todo esto sea posible se ha tenido que modificar el código Core del programa, la forma más sencilla era partir la imagen a renderizar por su eje Y, dicha partición genera N subpartes(siendo N el total de cores a nivel distribuido), cada procesador ejecuta una de las N subpartes.

Dado que el entorno de ejecución(servidores/laboratorios de la universidad) era bastante diverso en cuanto a arquitecturas de procesadores y sistemas operativos(así como versiones de las DLL asociadas) merece la pena destacar que se ha tenido que realizar una aproximación poco convencional para poder compilar los distintos ejecutables.



La aproximación “poco convencional” empleada para compilar ha sido la mostrada en el diagrama anterior, se hace una compilación distribuida, empleando 2 máquinas físicas que cuentan con diferentes arquitecturas de CPU, adicionalmente para evitar los problemas relacionados a las DLL, se simula por medio de Docker el SO operativo exacto del servidor donde se va a lanzar la simulación.

Máquinas	N máquinas	Arquitectura	SO	N Cores	RAM	N Cores Total	RAM Total
Berlin	1	x86	CentOS 9	32		32	
Pilgor	1	ARM	CentOS 8	96	318 GB	96	318 GB
Lab 102	20	x86	CentOS 8	6		120	
Lab 004	30	x86	CentOS 8	4		120	
TOTAL	52	-	-	-	-	368	...

### 3.3.2 BVH

Dado que el render final iba a estar construido a base de objetos renderizados en triángulos resultaba crucial optimizar esto dado que de no hacerlo esto repercutirá muy significativamente de forma negativa en el coste limitando el render final por una cuestión de tiempo.

La idea es la siguiente cada objeto construido a base de triángulos que se cargue, será almacenado en un [BVH\(Bounding volume hierarchy\)](#), cada BVH en tiempo de construcción calcula las dimensiones totales de su hitbox, a la hora de trazar los rayos en busca de colisiones se trazan contra la hitbox del BVH y no contra todos los triángulos que esta contiene, en caso de que si impacte en la hitbox puede ocurrir una de las dos siguientes situaciones: La primera es que el BVH está a su vez conformado por dos BVH por lo que se repite el proceso anterior de forma recursiva, la segunda opción es que nos encontramos en un BVH que no tenga más BVH en su interior en cuyo caso se realiza el trazado del rayo a **todas** las figuras que este tiene asignadas. La mejora que está optimización supone es inmensa.

### 3.3.3 LOD

Si seguimos tratando de optimizar los objetos renderizados a base de triángulos llegamos a la conclusión de que si un objeto está en un primer plano de la escena queremos visualizar este en la mayor calidad posible, no obstante si este se encuentra al fondo de la escena aunque se renderice en su máxima calidad los detalles no serán apreciados.

Esto se traduce en lo comúnmente denominado [LOD\(Level of detail\)](#), para implementar el mismo se ha realizado lo siguiente partiendo de los distintos modelos que van a ser renderizados con ayuda de blender se han generado múltiples instancias de estos cada uno con un nivel de detalle distinto, posteriormente en el código principal simplemente se debe elegir el objeto con el nivel de detalle concreto deseado.

### 3.3.4 Lazy Eval / Strict Eval

Una de las maravillas de Haskell como lenguaje de programación es la denominada evaluación perezosa, esta consiste en que a pesar de que un valor se define este no va a ser empleado hasta que su valor exacto sea necesitado por otro elemento del programa(que a su vez debe ser evaluado de otra forma como puede ser mostrarse por pantalla)

A continuación se adjunta un ejemplo:

```
A = 5  
B = A*A  
show B
```

Lo que haskell haría en este caso es ir acumulando los valores sin despejar(como si de una ecuación se tratase) en una expresión similar a un grafo, una vez alcanzado un punto de evaluación significativo(el show) se evaluaría dicha expresión(es decir hasta el show los valores reales de A y B se desconocen).

Esta mecánica tan particular trae consecuencias tanto positivas como negativas, por un lado si nos encontramos en una situación de incertidumbre donde no se sabe a priori si es necesario calcular un valor o no(por ejemplo intersectar el rayo con los triángulos en la BVH) esta forma de ejecutar el código nos puede suponer un ahorro muy significativo de cálculos lo cual se traduce en un menor tiempo de ejecución.

Por el otro lado dado que la expresión va creciendo hasta alcanzar un punto estricto de evaluación, implica que si concatenamos muchas manipulaciones/declaraciones de objetos en el código la expresión va aumentando su tamaño significativamente, así mismo está cada vez cuesta más tiempo dado que parte del tiempo de ejecución se tiene que emplear en manipular dicha expresión. Dicho de otra forma, si tenemos la certeza de que una serie de instrucciones/valores van a ser empleados, la evaluación perezosa laстра de forma innecesaria y significativa nuestro rendimiento tanto en tiempo como en memoria.

La solución son los denominados [Bang patterns](#), estos consisten en añadir un “!” en las asignaciones que deban ser evaluadas de forma estricta inmediatamente.

Es decir si en el código empleamos los bang patterns donde tenemos certeza de que se va a realizar la evaluación y además empleamos la mecánica de la evaluación perezosa en sitios

donde a priori no es seguro que sea necesario evaluarlos, obtenemos las mejoras de ambos paradigmas lo cual se traduce en mejoras muy significativas de rendimiento.

### 3.3.5 Optimización a nivel de compilado

Dado que Haskell es un lenguaje compilado esto implica la existencia de un compilador, por lo que podemos realizar ciertos trucos sobre sus flags así como sobre nuestro propio código para exprimir todo su rendimiento.

La mecánica más común en compiladores para aumentar el rendimiento es el flag “-O”, concretamente el compilador [GHC\(Glasgow Haskell Compiler\)](#) nos permite emplear hasta “-O2”. Otros flags que pueden resultar interesante son “subflags”, es decir flags que el compilador subyacente de C pueda emplear, tal vez el más relevante de estos sea el “-fast-math” este agiliza el procesado de datos de tipo float(perdiendo para ello algo de precisión).

Una vez tenemos los flags adecuados toca optimizar el código, la forma más sencilla de optimización consiste en declarar de forma explícita para el compilador que debe realizar *inlining* para ciertas funciones que van a ser llamadas **muchas** veces.

Si se aplica *inlining* el compilador reemplazará el código de llamada a la función por el propio código de la función, es decir en tiempo de ejecución nos vamos a ahorrar todo el proceso de salto de un punto arbitrario de código a la función así como la necesidad de realizar un cambio de contexto con las consecuencias pertinentes en pila.

Otros aspectos más propios de Haskell para optimizar el código, han sido el uso de listas por comprensión (la generación de estas por su naturaleza están más optimizadas en el código generado), el uso de multiwayif el cual realiza accesos más rápidos que un if normal ya que en cuenta de realizar las comprobaciones de forma secuencial emplea una tabla precalculada con las direcciones de salto.

Otro concepto propio de Haskell son las denominadas funciones Lambda, estas son funciones anónimas que no tienen un nombre asignado por lo que solo son llamadas en el lugar donde se encuentran definidas, esto implica un mayor rendimiento.

### 3.3.6 Pre Cálculo de estructuras

Dadas las consideraciones previas de mejoras de rendimiento, si por un momento se reflexiona acerca del funcionamiento del Photon Mapper se llega a la conclusión de que existen ciertas tareas repetitivas, que son repetidas de forma completamente innecesaria, por ejemplo la generación de BVHs, éstos se podrían recrear de 0 en cada una de las distintas instancias(a nivel de Core y de máquina) para recalcular siempre los mismos BVHs originales.

Es por ello que se ha optado por pre calcular todos los TAD(Tipo de Dato Abstracto) en una primera instancia, para posteriormente almacenar estos en un formato binario, de forma que no tengan que ser recalculados redundante e inútilmente en cada instancia del programa y sólo deban ser extraídos de los binarios generados previamente.

Se adjunta todo el código que lo implementa para ambos:

```
writeObject :: Binary a => FilePath -> DL.DList a -> IO ()
writeObject path obj = B.writeFile path (encode (DL.toList obj))

readObject :: Binary a => FilePath -> IO a
readObject = decodeFile
```

# Capítulo 4

## Resultados experimentales

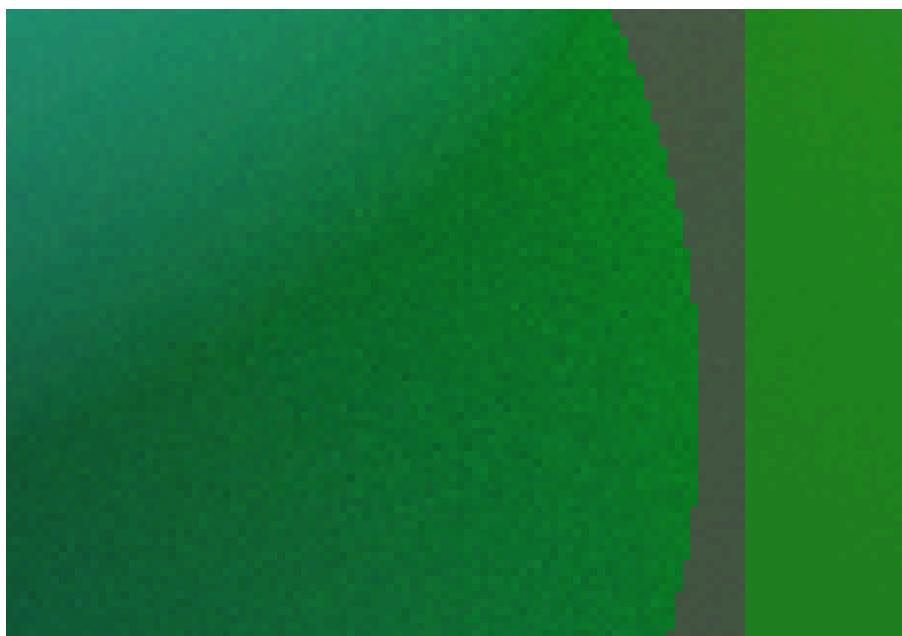
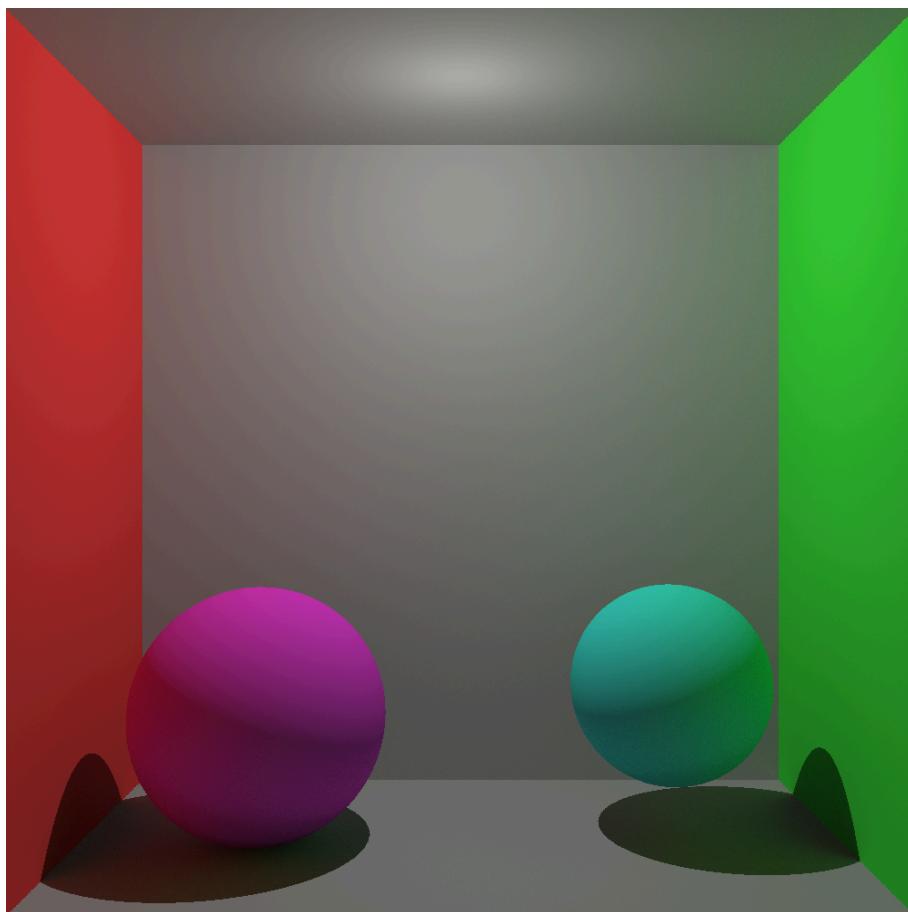
Todos los tiempos de referencia se realizan sobre una máquina con 10 cores de uso, no todos los que tenemos a nuestro alcance. Todas las imágenes se han sacado con 1000px y 200 smpp además de 256 ppp que esto nos indicará el nivel de detalle en la imagen.

Las siguientes muestras hacen referencia a las distintas propiedades que puede tener un objeto y se realizan, primero con una luz puntual y después con una luz de área que sería el techo en este caso.

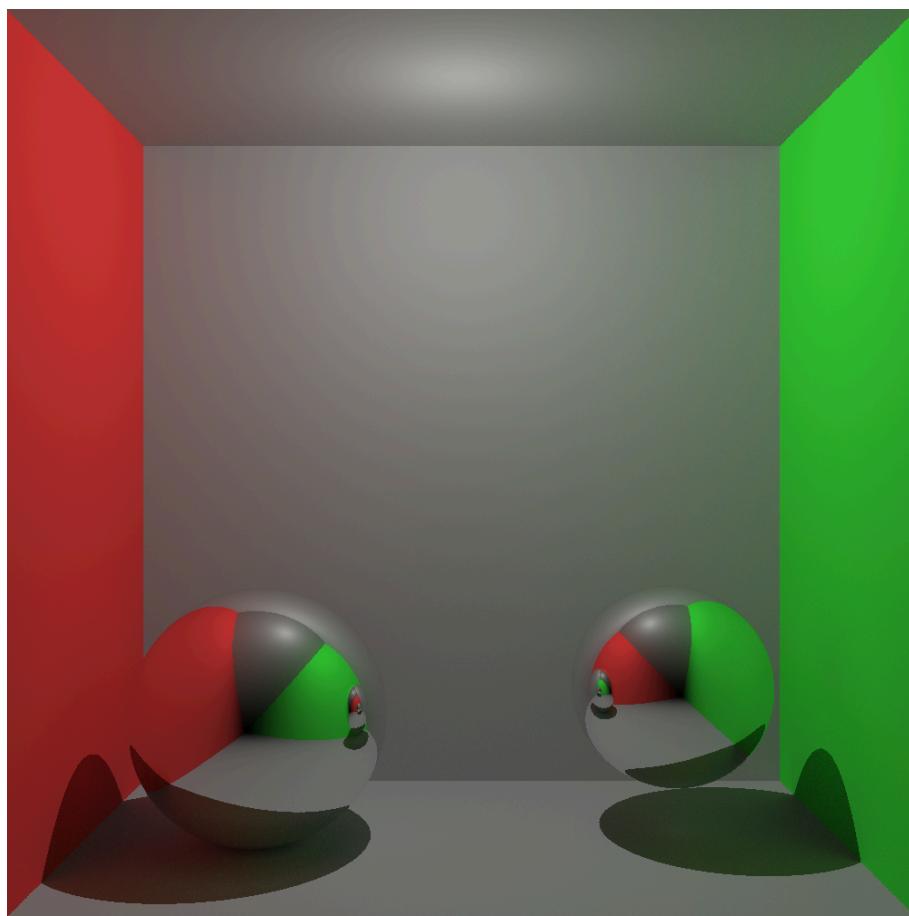
En la primera se realiza el muestreo uniforme del ángulo sólido y en luz de área se realiza el muestreo del coseno.

Otro detalle interesante, es que las imágenes obtenidas con luz puntual han tardado alrededor de 2200 segundos, en cambio la luz de área ha tardado en torno a los 350 segundos.

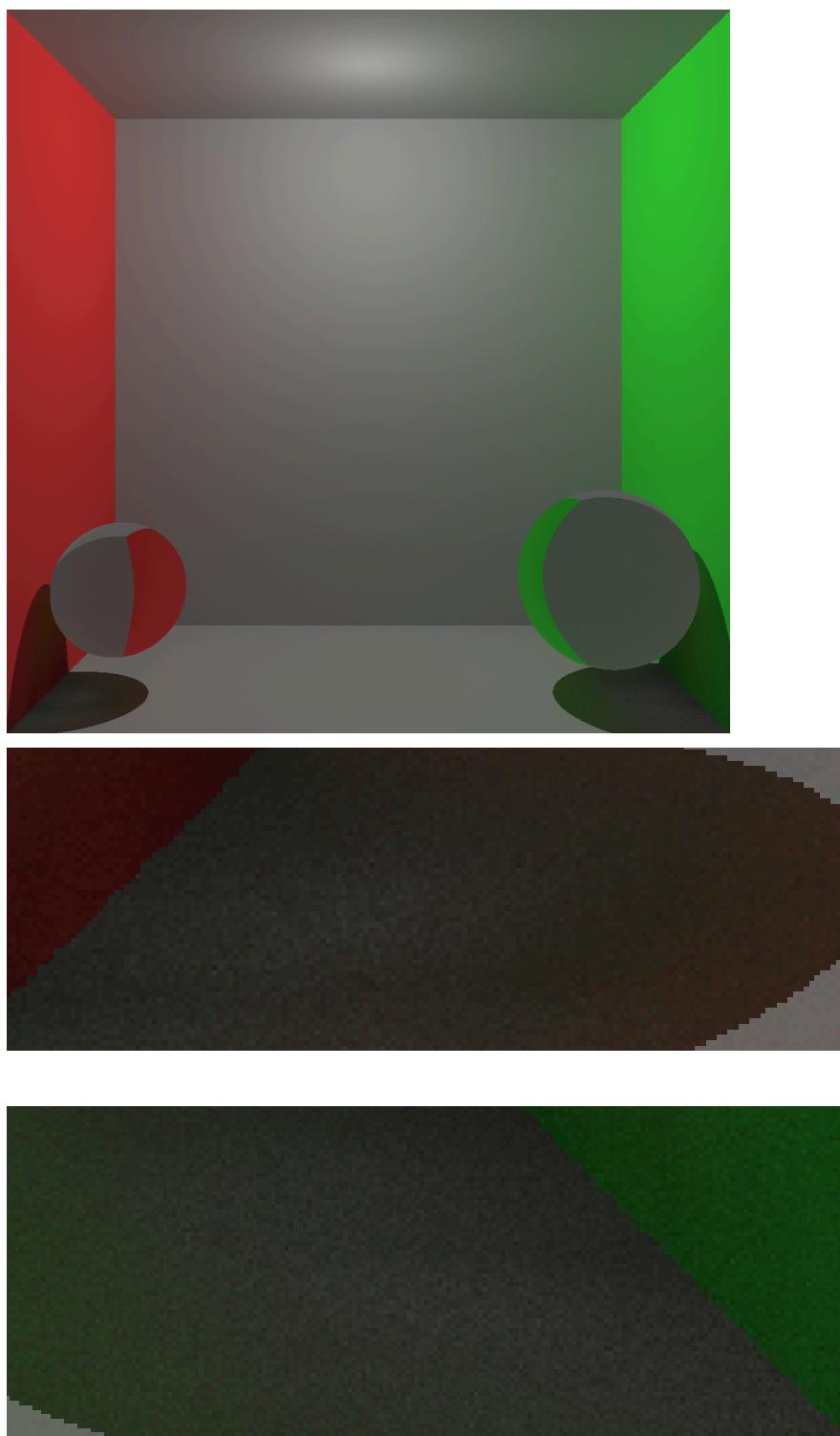
Este motivo se debe principalmente a dos factores, el primero de ellos es que en la luz puntual se hace uso del next-event-estimation, donde en cada iteración necesitas lanzar un rayo hacia la luz que es el que le aportará de color al objeto, en cambio en la luz de área esto no es necesario. Otro motivo es, que con luz puntual podría ser casi infinito si se ponen mal los parámetros, ya que el recorrido de un rayo acaba simplemente si se pierde el rayo o si se absorbe por las propiedades del material, así que si haces una habitación cerrada y todos los materiales con suma de tripletas a prácticamente 1 el recorrido de un simple rayo es infinito. En cambio con luz de Área encontramos otra directiva que nos obliga a parar la cual es chocar contra el objeto que emite la luz, así que, como en la mayoría de casos, chocas antes con un objeto que tenga luz que esperas a que se absorba o pierda un rayo y no tienes que calcular next-event-estimation, la luz de área tarda una 6 veces menos.



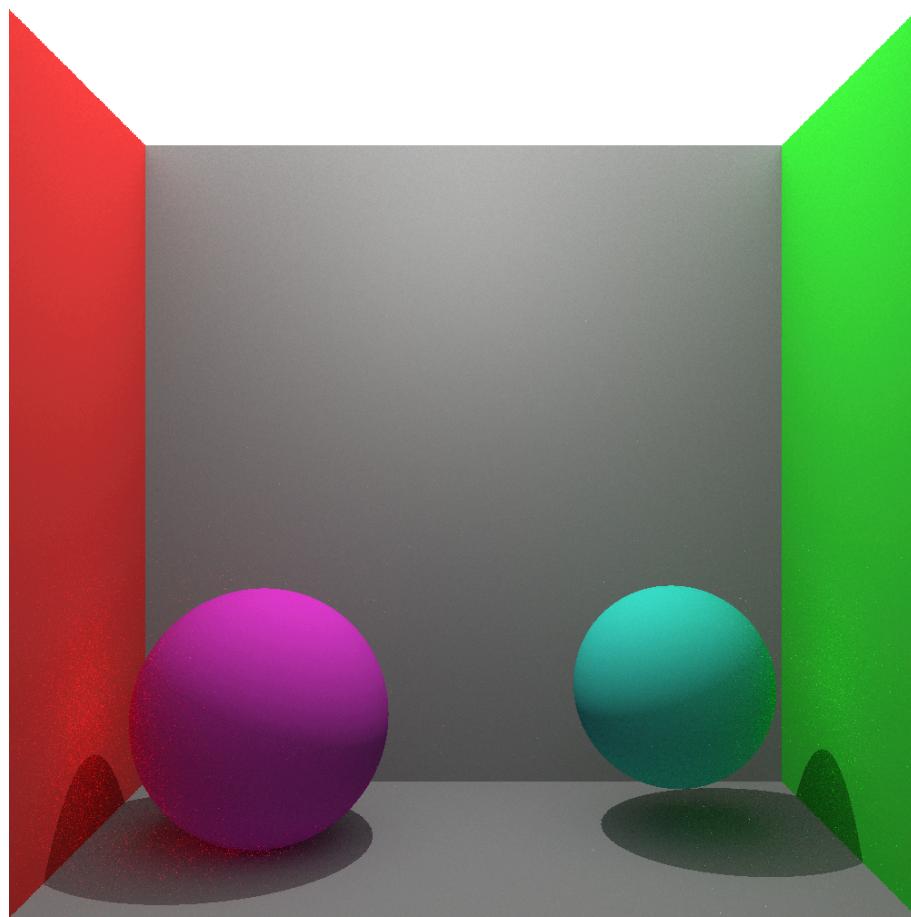
Con luz puntual y objetos difusos, si realizamos mucho zoom por ejemplo, entre la esfera y la pared podemos notar como la pared le aporta de color a la esfera. Esto también se da porque la esfera tiene una componente verde en su RGB, si esta no la tuviera este efecto no se llegaría a conseguir.



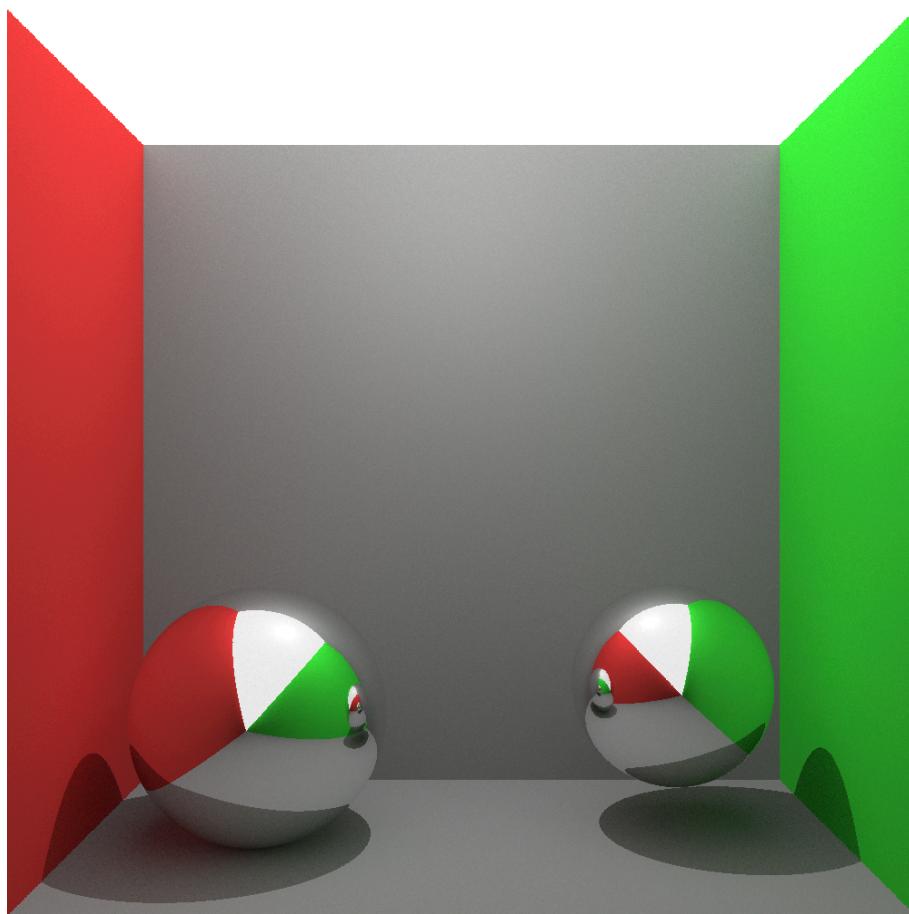
Con espejos con componente especular es sumamente importante lo comentado anterior de, a la hora de darle propiedades especulares en su tripleta, dejar un margen, por ejemplo  $ke = 0.9$ . Ya que si fuera == 1 y coinciden, estarían rebotando los rayos entre los espejos infinitamente.



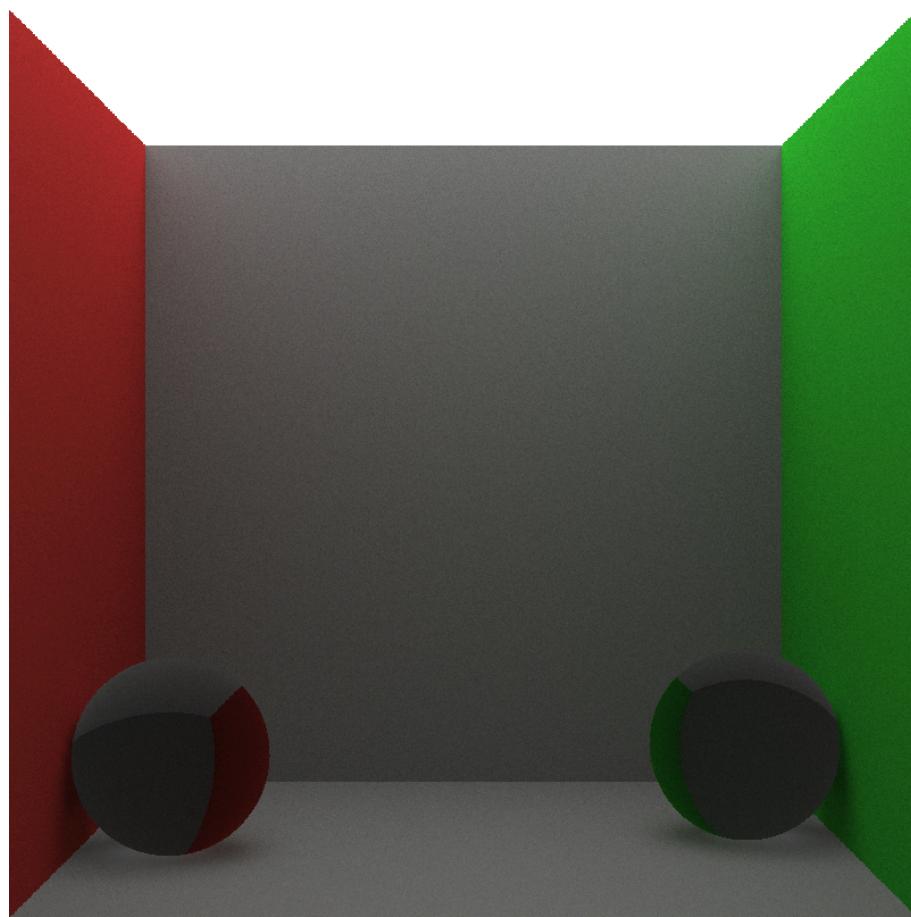
Con luz puntual y objetos refractantes que simulan el cristal encontramos pequeñas cáusticas en su base, de normal y por lógica deberíamos encontrar cáusticas, pero al realizar el next-event-estimation debe tocar exactamente el punto que miramos con dirección hacia el punto central de la luz, sin embargo, como se refracta el rayo esta nunca llega a impactar, a no ser de un caso muy específico, así las cáusticas que se generan es por luz indirecta, probablemente del techo que hay puntos donde más se concentra la luz.



Con luz de área en esferas difusas encontramos los efectos de usar el muestreo del coseno, donde, como no le aplicamos el término del coseno en la educación de render y la mayoría de los rayos se dirigen a la normal hay partes más quemadas de la escena.



De esta imagen no habria que decir nada que no se haya dicho en las anteriores, en esta no se quema a diferencia de la anterior porque la dirección de salida del rayo es la de un rayo especular y no hay que multiplicarlo por la probabilidad de salida del rayo.



A diferencia de con luz puntual, esta imagen sí que presenta algo más cáusticas debajo de las esferas ya que existen más caminos que llegan hasta la luz de área y este aporta más energía a la caustica.

# Capítulo 5

## Conclusiones

Tras analizar en profundidad todo el trabajo realizado, se llega a la conclusión de que se han obtenido unos resultados realmente buenos, teniendo el proyecto final un calibre bastante superior al exigido por la asignatura.

La elección inicial de implementar el Path Tracer en el lenguaje de programación Haskell, ha demostrado ser una muy buena decisión dado que esto ha repercutido enormemente en la agilización del desarrollo así como en la legibilidad y simplicidad del código implementado.

Respecto a los opcionales, se han alcanzado algunos opcionales los cuales debido a su nivel son objeto de estudio en el Máster Universitario en Robótica, Gráficos y Visión por Computador, los dos apartados extras más representativos de esto son las Ecuaciones de Fresnel y los Medios Participativos.

En cuanto a rendimiento se refiere, se ha logrado llevar al máximo el potencial del lenguaje de programación empleado, también se han tenido en cuenta consideraciones relativas puramente al mundo de la informática gráfica o de los videojuegos. Para brindar un poder de cómputo a la altura del resto del proyecto se ha recurrido a una implementación distribuida y concurrente.

Como era de esperar los resultados experimentales demuestran todo lo previamente confirmado, remarcando así la calidad del trabajo realizado previamente.

En conclusión se han alcanzado y sobrepasado con creces todas las metas propuestas en la asignatura inicialmente.

Como última conclusión y reflexión personal se adjunta la siguiente cita:

“Independientemente de la nota que reciba este proyecto, somos los primeros en toda la historia del grado en Ingeniería Informática en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza, que logran de forma exitosa y sobrepasando todas las expectativas y prejuicios, implementar el Path Tracer de una forma funcional pura.”

# Bibliografia

<https://www.haskell.org/>

<https://hackage.haskell.org/package/kdt>

[https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore\\_intersection\\_algorithm](https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm)

[https://www.arnoldrenderer.com/research/egsr2012\\_volume.pdf](https://www.arnoldrenderer.com/research/egsr2012_volume.pdf)

[https://es.wikipedia.org/wiki/Funci%C3%B3n\\_gaussiana](https://es.wikipedia.org/wiki/Funci%C3%B3n_gaussiana)

[https://en.wikipedia.org/wiki/Fresnel\\_equations](https://en.wikipedia.org/wiki/Fresnel_equations)

[https://en.wikipedia.org/wiki/Finite\\_difference](https://en.wikipedia.org/wiki/Finite_difference)

<https://ics.uci.edu/~majumder/VC/classes/BEmap.pdf>

<https://render-haskell.duckdns.org/>

<https://wiki.haskell.org/Concurrency>

<https://wiki.haskell.org/Parallel>

[https://en.wikipedia.org/wiki/Bounding\\_volume\\_hierarchy](https://en.wikipedia.org/wiki/Bounding_volume_hierarchy)

[https://en.wikipedia.org/wiki/Level\\_of\\_detail\\_\(computer\\_graphics\)](https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics))

[https://downloads.haskell.org/~ghc/7.8.4/docs/html/users\\_guide/bang-patterns.html](https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/bang-patterns.html)

<https://www.haskell.org/ghc/>