

# Photon Mapper

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 Construcción del KDT	4
2.2 Ecuación de render	5
<b>Capítulo 3</b>	<b>6</b>
<b>Implementación del algoritmo</b>	<b>6</b>
3.1 KDT	6
3.2 Ecuación de render	9
3.3 Implementación de las extensiones	10
3.3.1 Geometrías adicionales	10
3.3.1.1 Rectángulo	10
3.3.1.2 Triángulo	11
3.3.1.3 Mallas de triángulos - OBJ	11
3.3.2 BRDF de Phong	12
3.3.3 Carga de objetos 3D	13
3.3.4 Formatos de salida adicionales	14
3.3.5 Medios participativos	15
3.3.6 Utilización de distintos Kernels y modos	17
3.3.7 Texturas	18
3.3.8 Ecuaciones de Fresnel	20
3.3.9 Bump Mapping	21
3.3.10 Documentación profesional	22
3.4 Mejoras de rendimiento	23
3.4.1 Concurrencia y distribución	23
3.4.2 BVH	25
3.4.3 LOD	25
3.4.4 Lazy Eval / Strict Eval	26
3.4.5 Optimización a nivel de compilado	27
3.4.6 Pre Cálculo de estructuras	28
<b>Capítulo 4</b>	<b>29</b>
<b>Resultados experimentales</b>	<b>29</b>
<b>Capítulo 5</b>	<b>34</b>
<b>Conclusiones</b>	<b>34</b>
<b>Bibliografía</b>	<b>35</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 Photon Mapping para lo relativo a la simulación de la luz de una forma más precisa, permitiendo así que se generen unas mejores cáusticas y otros efectos de la luz que con el Path tracer no se pueden llegar a lograr. Una vez comprendida la base teórica se han aplicado estos conocimientos programando nuestro propio generador de renders basado en Photon Mapping.

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.

La base teórica de la técnica de Photon Mapping es la siguiente, si se quiere generar una imagen con cierta clase de elementos avanzados tales como por ejemplo elementos que reflejan o refractan la luz, surge el siguiente problema: si los rayos son lanzados desde la cámara y pasan por un objeto de estas características es muy poco probable que los rayos rebotados sean capaces de encontrar una luz puntual, lo cual se aprecia en la generación de cáusticas del Path Tracer. Para solventar dicho problema primero se lanzan “fotones” emitidos de forma aleatoria por las fuentes de luz y estos se almacenan en una estructura(se detalla más adelante) para que, posteriormente, cuando se lancen los rayos desde la cámara, se puedan consultar los “fotones” almacenados sin la necesidad de trazar un nuevo rayo hacia la luz, esto se traduce en un aumento muy significativo en los efectos causados por la luz.

# Capítulo 2

## Diseño del algoritmo

### 2.1 Construcción del KDT

La creación de la estructura de datos que almacenará los fotones es de lo más importante del Photon Mapping. Para esta estructura se ha optado por trabajar con un KDT el cual es una estructura de datos que partitiona el espacio para organizar los puntos en un Espacio euclídeo de  $k$  dimensiones. Para la creación de este hay que tener en cuenta primero el número de luces de la escena y su potencia, además de, en nuestro caso, el número total de fotones que emitirán las luces. Así indicamos que lanza más fotones aquella luz con una mayor potencia.

Otra particularidad que viene dada por el lenguaje empleado es que al no existir los bucles en este, todo se hace por medio de recursividad.

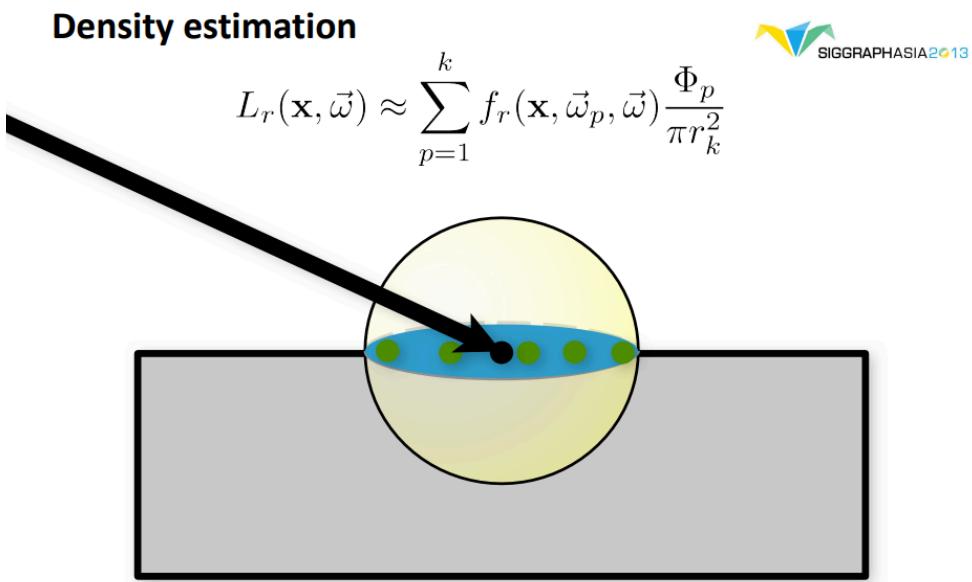
El algoritmo conceptualmente consiste de varias partes, primero, si quedan fotones pendientes de lanzar, se lanzan, si por el número de fotones lanzados toca cambiar de luz, se cambia, y si no, la luz actual elige una dirección aleatoria y lanza el fotón.

Aquí se entra en otra recursividad. Primero se comprueba si puede rebotar más el fotón o si se ha perdido (no ha chocado contra nada), sino es se lanza la ruleta rusa con el objeto al que se ha golpeado, así se comprueba que hacer, si se absorbe el fotón entonces se corta la recursividad, si se obtiene difuso, entonces guarda el nuevo fotón y lanza otro aleatorio y si se obtiene especular o reflectante, se sigue la trayectoria del fotón.

Una vez se completan todos los bucles recursivos resulta la lista final de fotones, la lista final es almacenada en un archivo binario (esto se hace para ahorrar en cálculos innecesarios debido a que en todas las instancias, del programa en concurrente y distribuido, el render tendrá el mismo árbol), y a la hora de crear el render será cargado en el kdt para su posterior uso.

## 2.2 Ecuación de render

A la hora de resolver la ecuación de render no es como en el Path Tracer, donde consistía en mirar en la semiesfera la luz recibida en ese punto y como afecta al objeto. Ahora al tener los fotones por la escena, cuando un rayo de la cámara golpea un objeto, en vez de resolver la ecuación de render, el programa cogerá o los k-Fotones más cercanos o todos los fotones en el radio que se le indique. Así la nueva ecuación para estimar la radiancia en un punto quedaría de la siguiente manera:



Donde la radiancia en el punto  $\mathbf{x}$  se puede igualar al sumatorio de los  $k$  fotones, con la brdf del objeto multiplicada por un kernel, en la imagen se presenta un kernel constante donde le da el mismo peso a todos los fotones y solo multiplica el flujo de los fotones entre el radio buscado al cuadrado multiplicado por pi.

Algo relevante a la hora de procesar los fotones es que solo se tienen en cuenta los fotones que están en el propio objeto. Dicho de otro modo, primero se hallan los fotones más cercanos y después se les aplica un filtro para solo mirar los fotones que han chocado en el propio objeto, este parámetro se guarda cuando se genera la lista de fotones que se convertirá posteriormente en el kdt.

La parte de los kernels se trata con una mayor profundidad más adelante, ya que, además del constante se ha desarrollado alguno más opcional.

Esto es así para los materiales perfectamente difusos donde solo importan sus fotones, en cambio en materiales especulares, refractantes, o que presentan una mezcla de diversas propiedades, se ha optado por mirar sus diversos componentes a la vez. Así, si es un material por ejemplo difuso y especular, se comprobarán los fotones del propio objeto y a esos se le sumarán los fotones del objeto reflejado.

# Capítulo 3

## Implementación del algoritmo

### 3.1 KDT

Dado que no se ha empleado el lenguaje de programación **recomendado** por el profesorado de la asignatura, el código facilitado por dicho profesorado que contiene una implementación de KDT con el lenguaje de programación C++ evidentemente ha resultado inútil para el código desarrollado en Haskell, es por ello que para subsanar esta carencia se ha optado por emplear una implementación existente del KDT disponible en una de las librerías estándar de Haskell. La librería interna de Haskell empleada es [kdt](#), dicha librería ofrece dos implementaciones distintas del kdt una estática y una dinámica, teniendo en cuenta que todos los fotones son generados de golpe, se ha decidido emplear la implementación estática, la cual solo permite añadir fotones al kdt durante la creación del mismo y no a posteriori, esta decisión repercute de forma positiva en el rendimiento.

Para seguir unas buenas prácticas de programación se ha creado el código que maneja el KDT en un módulo aparte, a continuación se adjunta [todo](#) el código de dicho módulo.

```
{-# LANGUAGE RecordWildCards #-}
module KdTest where
import Data.KdTree.Static (build, KdTree)
import ELEM3D ( Foton(..), Point3D(..) )

{-# INLINE fotonAxis #-}
fotonAxis :: Foton -> [Float]
fotonAxis (Foton {pFot = Point3D x y z}) = [x, y, z]

{-# INLINE createKD #-}
createKD :: [Foton] -> KdTree Float Foton
createKD = Data.KdTree.Static.build fotonAxis
```

Para la parte de la creación de la lista de fotones se ha empleado la implementación descrita en el apartado de diseño del algoritmo.

La primera parte hace referencia a los fotones que salen directamente de la luz, donde tiene que tener en cuenta, la luz actual, su potencia y también se decide el rayo de salida

```
createPhoton :: Float -> DL.DList Foton -> Int -> Int -> Set.Set Shape -> [Luz] -> StdGen
-> Int -> DL.DList Foton
createPhoton lzT fotones contador contMx figuras luces gen nRebotes
| contador == contMx = fotones
| contador == contMx `div` round (lzT / intLuz) = createPhoton (lzT - intLuz) fotones
                                                (contador+1) contMx figuras (tail luces) gen' nRebotes
| otherwise = createPhoton lzT newlisP (contador+1) contMx figuras luces gen' nRebotes
where
  (ray, Luz pointPapa rgbPadre intLuz) = selecLightSource luces contador contMx gen
  !newlisP = traceRay pointPapa ((4.0 * pi * intLuz) / fromIntegral contMx) rgbPadre
            fotones figuras nRebotes gen' nxtObj
  nxtObj = obtenerPrimeraColision $ Set.map (\figura -> oneCollision figura ray)
            figuras
  gen' = snd $ split gen
```

Una vez el fotón es lanzado desde la luz, la función entra en el siguiente bucle recursivo para ver cual es el comportamiento del fotón. Principalmente el código diferencia la interacción del fotón según la ruleta rusa, esta decide cómo se comporta el fotón con respecto al objeto, si se absorbe, o si interactúa como objeto difuso, refractante o especular.

Solo se almacenará el fotón siempre y cuando la Ruleta Rusa nos devuelva “Difuso” (esto se hará con la función “*fotones `DL.snoc `foton`*”).

Cabe destacar que en la primera implementación se empleaban las funciones básicas de haskell de concatenar un elemento a una lista (“*fotones ++ [foton]*”) sin embargo esta operación tiene de coste O(n) con ‘n’ la longitud de la lista. En cambio definiendo la lista de fotones como una DList de fotones esta operación pasa a tener tiempo constante.

Con el primer método, solo creando una lista de aproximadamente 16\_000 fotones, tenía un coste temporal de 4 segundos, con el método alternativo ahora en ese tiempo es posible generar más de un 1\_700\_000 fotones.

```

traceRay :: Point3D -> Float -> RGB -> DL.DList Foton -> Set.Set Shape -> Int -> StdGen ->
Obj -> DL.DList Foton
traceRay p pot rgb fotones figuras n gen obj
| n == 0 || rgb == RGB 0 0 0 = fotones
| otherwise = result
where
  result = case caso of
    0 -> photonD -- Difuso
    1 -> photonR --Refracción
    2 -> photonE -- Especular
    _ -> fotones -- Absorción

  pObj = colObj obj
  nObj = normObj obj

  (caso, por) = ruletaRusa (trObj obj) gen --RULETA RUSA

  photonD = traceRay pObj (2*pi*pot'*por* abs (w0Obj nxtObj .* nObj)) (brdf obj
    figuras) (fotones `DL.snoc` foton) figuras (n-1) gen' nxtObj
  photonE = traceRay pObj (pot * por) (brdf obj figuras * rgb ) fotones figuras n gen'
    objEsp
  photonR = traceRay pObj (pot * por) (brdf obj figuras * rgb ) fotones figuras n gen'
    objCri

  foton = Foton pObj pot' rgb (idObj obj) -- Definir el nuevo fotón
  pot' = abs (w0Obj obj .* nObj)*pot / ((1+(modd (colObj obj #< p)/10.0))**2)
  -- Nueva potencia del fotón después de chocar con un objeto

  nxtObj = objAleatorio figuras obj gen -- Siguiente objeto que choca con dir random
  objEsp = objEspejo figuras (w0Obj obj) nObj pObj -- Siguiente obj choca con dir esp
  (objCri, _) = objCristal figuras (w0Obj obj) nObj 1 (reflObj obj) pObj
  -- Siguiente objeto que choca con dirección refracción
  gen' = snd $ split gen

```

## 3.2 Ecuación de render

Como se ha descrito en el apartado de diseño del algoritmo de la ecuación de render, una vez se realiza la colisión del objeto se consultan las 3 propiedades del objeto y para cada una se realiza su densidad de estimación. Es obligatorio realizar a priori un filtrado previo de los fotones para que solo se empleen los del propio objeto, la parte previa a la estimación de densidad es la siguiente:

```
kdToRGB :: KdTree Float Foton -> Float -> Set.Set Shape -> Obj -> RGB
kdToRGB kdt 0 figuras obj = RGB 0 0 0
kdToRGB kdt radio figuras obj = newRGB
  where

    photons = inRadius kdt radio (pointToPothon (colObj obj))
    -- photons = kNearest kdt (round radio) (pointToPothon (colObj obj))
-- Si quisieramos coger los k-fotones más cercanos
    photons' = filter (\fot -> idObj obj == idFot fot) photons
-- Coger solo fotones del mismo objeto

    !newRGB = if null photons' then RGB 0 0 0 else estDensPhoton photons' obj figuras radio
-- Si no hay fotones el color es 0, sino ecuación de estimación de densidad
```

Aquí está descrito para utilizar los fotones en el radio dicho, para cambiarlo para coger los ‘k’ fotones más cercanos se debería emplear la línea de abajo que está comentada.

La estimación de densidad después, queda de la siguiente manera:

```
estDensPhoton :: [Foton] -> Obj -> Set.Set Shape -> Float -> RGB
estDensPhoton photons obj figuras radio = newRGB
  where
    -- newRGB = sumRGB $ map (\photon -> fusion obj (fGaus photons obj photon) photon)
    photons
    -- newRGB = sumRGB $ map (\photon -> fusion obj (1/(radio * radio * pi)) photon) photons
    newRGB = sumRGB $ map (\photon -> fusion obj (1 / (1 + distFot (colObj obj) photon))) photon) photons
    fusion :: Obj -> Float -> Foton -> RGB
    fusion obj kernel fot = newRGB `modRGB` kernel
      where
        newRGB = modRGB (scale $ rgbFot fot) (iFot fot) * brdf obj figuras
```

En resumen, se realiza el sumatorio de la aportación de cada fotón en la función local “fusion”, donde “kernel” es una variable Float de la que se hablará más a detalle posteriormente.

## 3.3 Implementación de las extensiones

### 3.3.1 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:

#### 3.3.1.1 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
```

### 3.3.1.2 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](#).

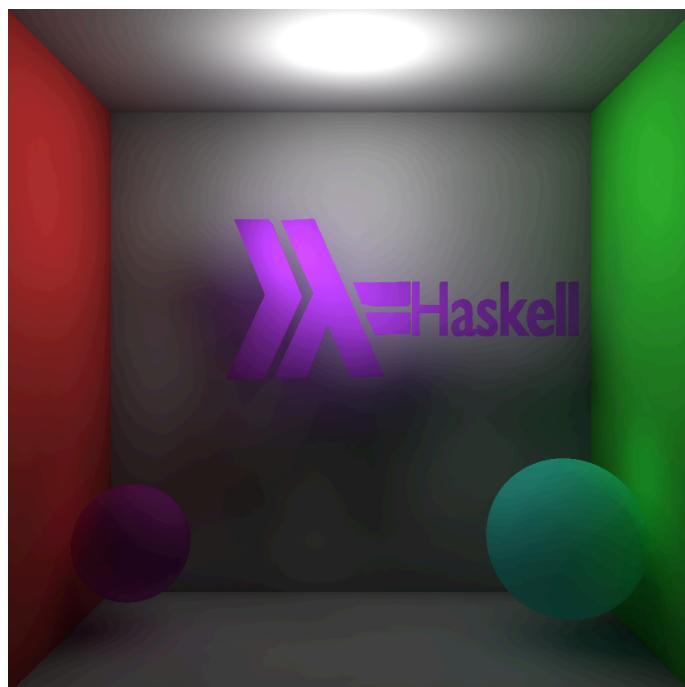
El siguiente código implementa la colisión de un rayo con un triángulo:

```
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)
```

### 3.3.1.3 Mallas de triángulos - OBJ

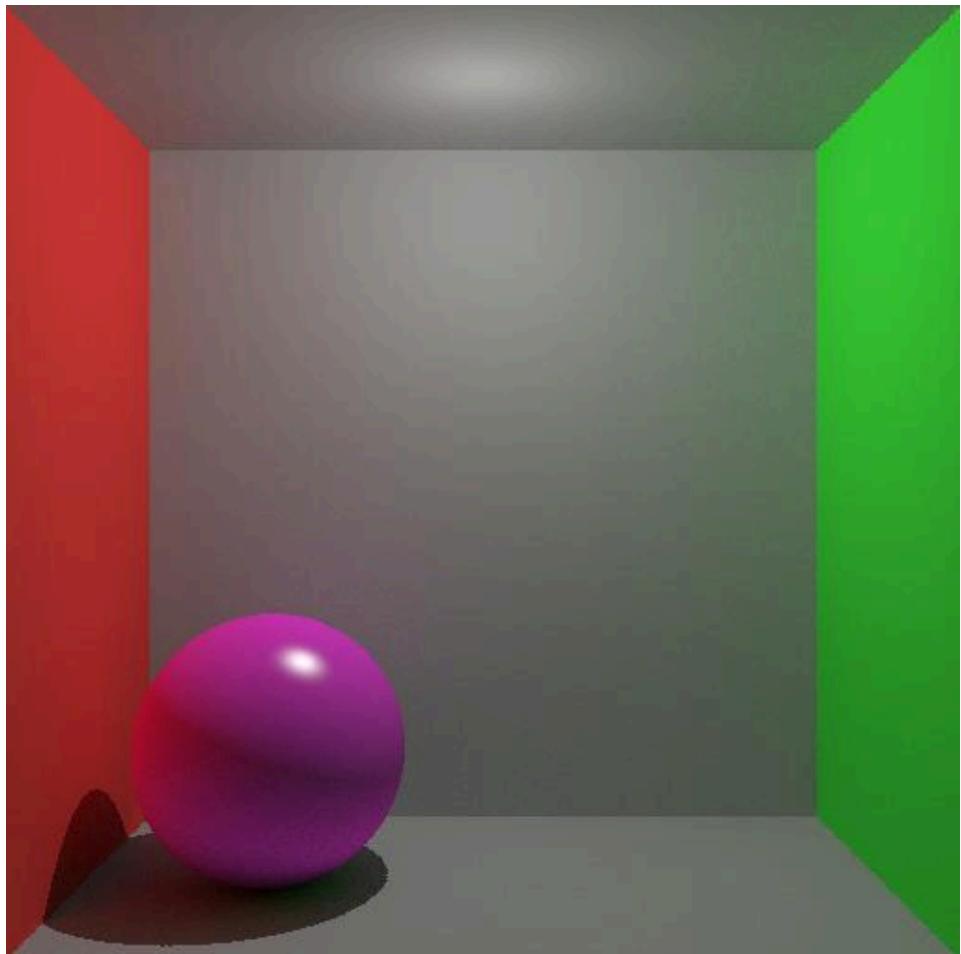
Dado que un obj contiene una gran cantidad de triángulos que conforman un mismo objeto, se ha decidido tratar estos como tal, para ello se define la geometría de la malla de triángulos (esta definición va estrechamente ligada a la optimización asociada a la mism BVH). Para la carga de esta geometría 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.



### 3.3.2 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
```

### 3.3.3 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'')
```

### 3.3.4 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
    ]
```

### 3.3.5 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*

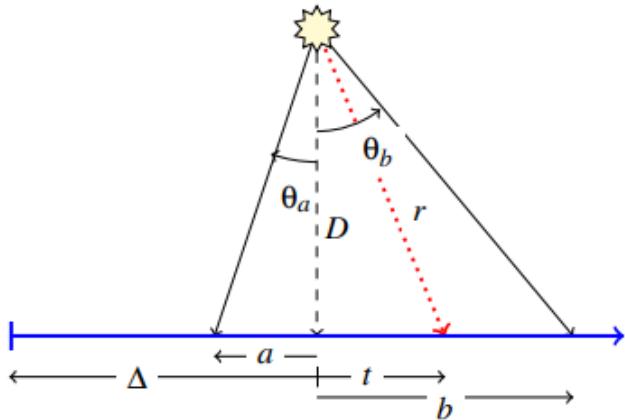
$$\begin{aligned}
 dL(\mathbf{x}, \omega) &= -\mu_t(\mathbf{x}, \omega)L(\mathbf{x}, \omega) dz && \text{>> Assuming extinction only} \\
 \frac{dL(\mathbf{x}, \omega)}{L(\mathbf{x}, \omega)} &= -\mu_t(\mathbf{x}, \omega) dz && \text{>> Reorganizing } dz \\
 \ln(L(\mathbf{x}_z, \omega)) - \ln(L(\mathbf{x}_0,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

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



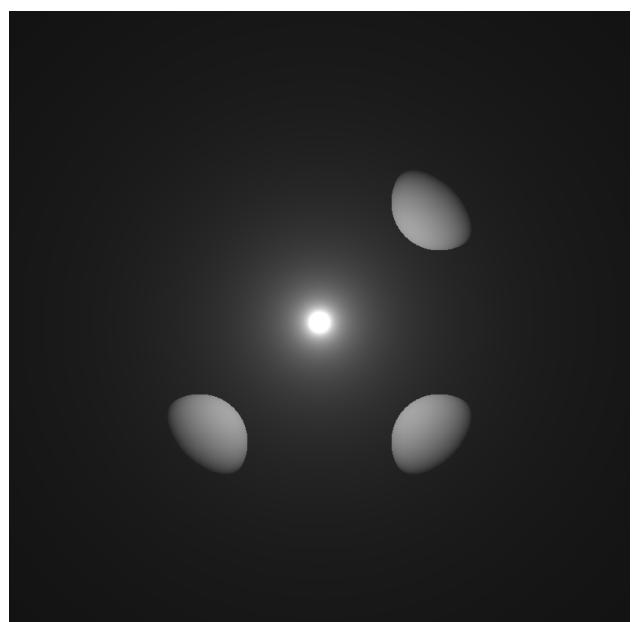
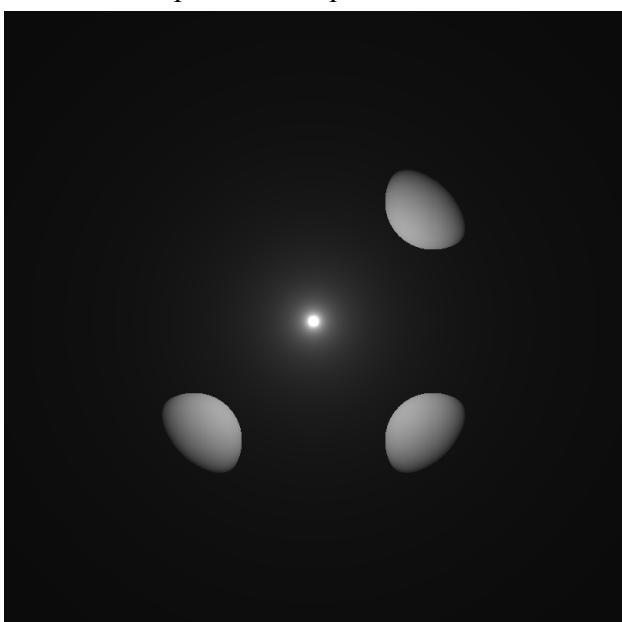
Sin embargo, como la luz renderizada es total se tuvo que tomar una aproximación distinta, esta consiste en que, para añadirle el efecto de niebla, en el rayo trazado al objeto, se halla el punto más cercano a la luz, se calcula la distancia a esta, y se eleva a -2.

```

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.



### 3.3.6 Utilización de distintos Kernels y modos

Los kernel del photon mapping hacen referencia a cuánto peso tienen los fotones que se han tenido en cuenta para la estimación de densidad, el más sencillo de implementar es un kernel constante donde todos los fotones tienen el mismo peso, no obstante es del todo correcto el hacerlo de esta manera.

Otros kernels que se han implementado y que aportan una mayor fidelidad física a la ecuación de render original, estos son el kernel de distancia, y el de distancia al cuadrado donde, cuanto más cerca del centro de búsqueda esté, más peso tienen, y cuanto más lejos menos, por otro lado el de distancia normal aporta un peso que se reduce linealmente cuanto más lejos está el fotón del centro, el de distancia al cuadrado se reduce de manera cuadrática.

Por último, se ha decidido a su vez implementar un Kernel Gaussiano. Este tiene en cuenta todos los fotones recolectados y aplica una [función Gaussiana](#) para asignarles sus pesos.

Además, con la facilidad del kdt en Haskell se han probado los dos modos de coger los fotones, ya sean en un radio dado y coger los ‘k’ más cercanos.

Las diferentes pruebas y resultados de este apartado se encuentran en el capítulo 4.

### 3.3.7 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
```

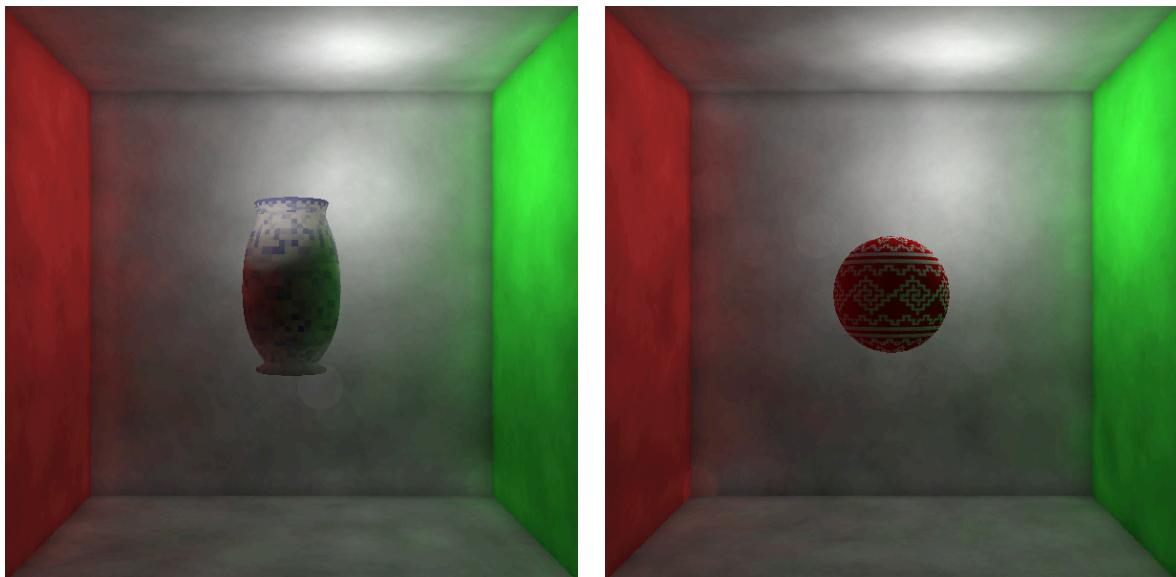
Para la esfera es necesario conocer los puntos polares (latitud y longitud) en los que se ha colisionado el punto, con esto, ya se puede hallar el u y el v.

```
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.



### 3.3.8 Ecuaciones de Fresnel

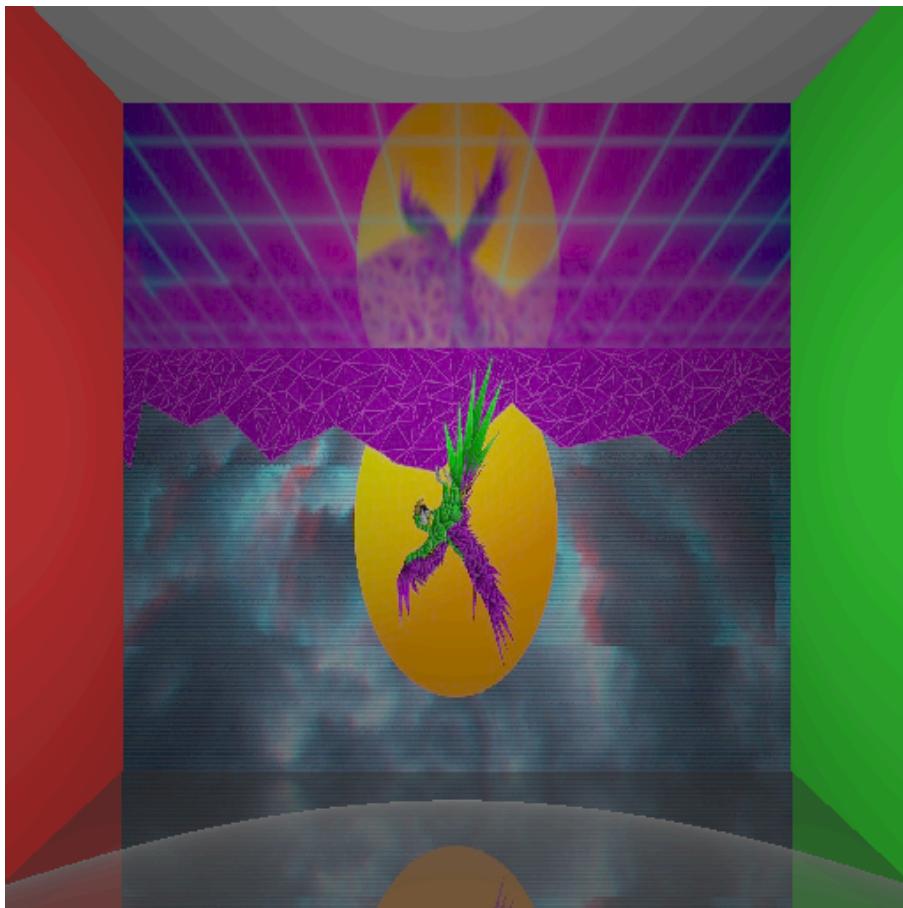
Para sacar la siguiente imagen se han modificado las propiedades de la cámara para poder medir bien su efecto sobre un suelo con [ecuaciones de fresnel](#) implementadas para que se vea el efecto, las ecuaciones son las siguientes:

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.

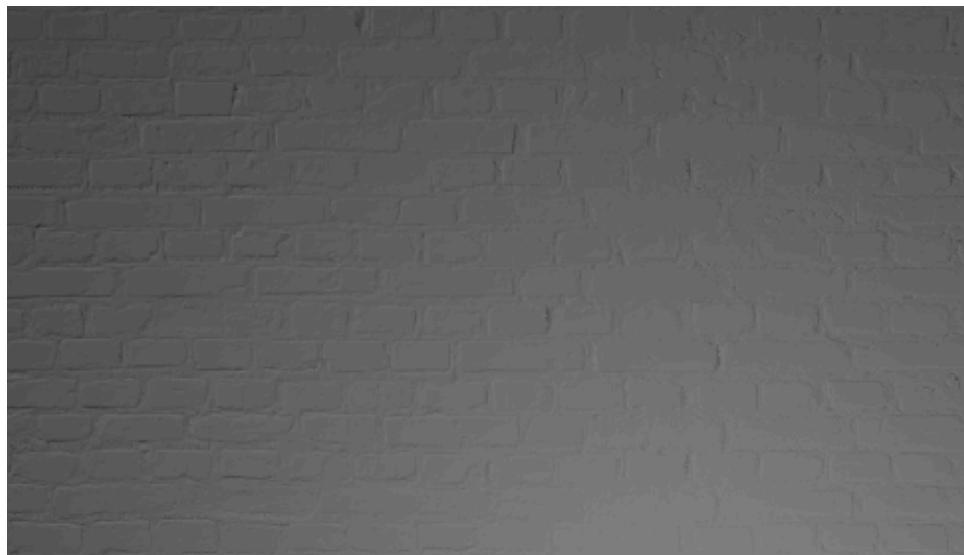
### 3.3.9 Bump Mapping

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](#).

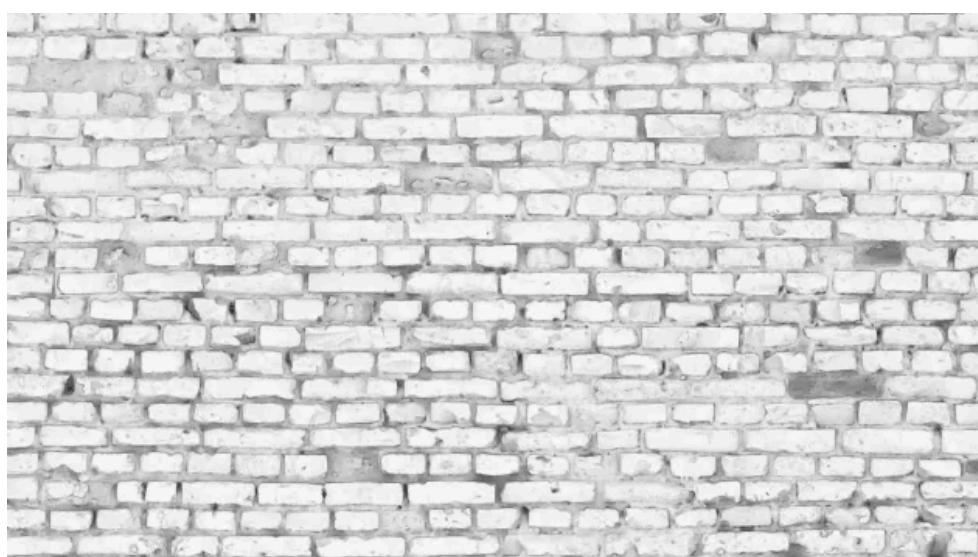
En resumidas cuentas se indica que, la nueva dirección de la normal, será vista desde un cambio de base sobre la normal de la figura, la Dirección( $B_u, 1, B_v$ ) normalizada. Los parámetros  $B_u$  y  $B_v$  se extraen de esas diferencias finitas, donde  $B_u$  es el valor del pixel, pasado de RGB 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:



### 3.3.10 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.

## 3.4 Mejoras de rendimiento

### 3.4.1 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.

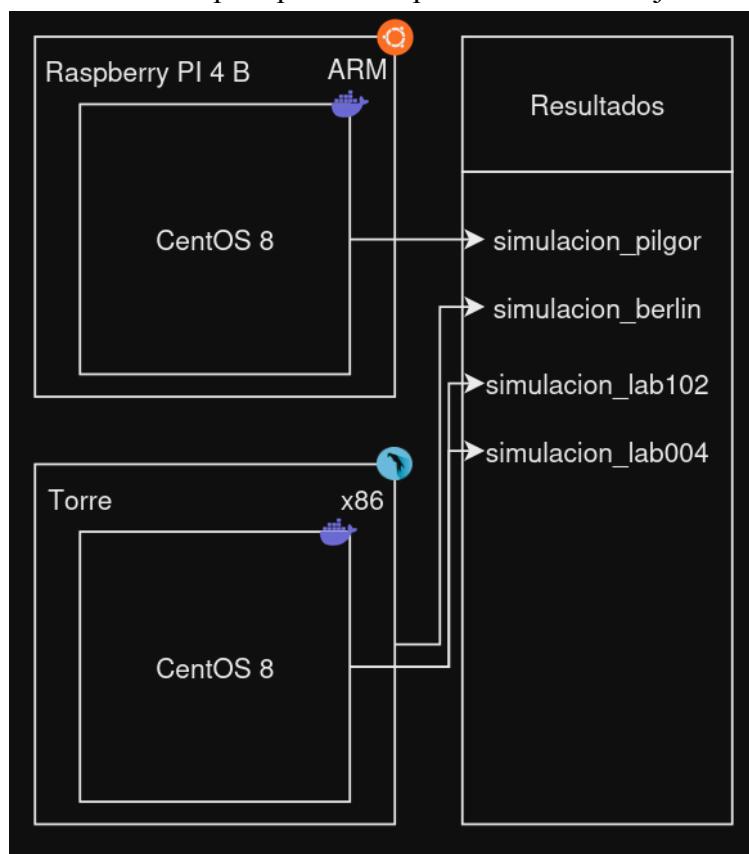


De esta forma primero tenemos el core programado en Haskell el cual acepta parámetros de ejecución, en la capa intermedia nos encontramos con un script de Bash el cual gestiona la concurrencia(emplea para ello procesos asíncronos y barreras), este script es el encargado de

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	...

### 3.4.2 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.

### 3.4.3 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.

### 3.4.4 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.

### 3.4.5 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.

### 3.4.6 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, el ejemplo más claro es la generación del KDT, este se podría recrear de 0 en cada una de las distintas instancias(a nivel de Core y de máquina) para recalcular siempre el mismo KDT original.

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.

Esto se ha aplicado sobre todo con los TAD del KDT y los BVH.

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))

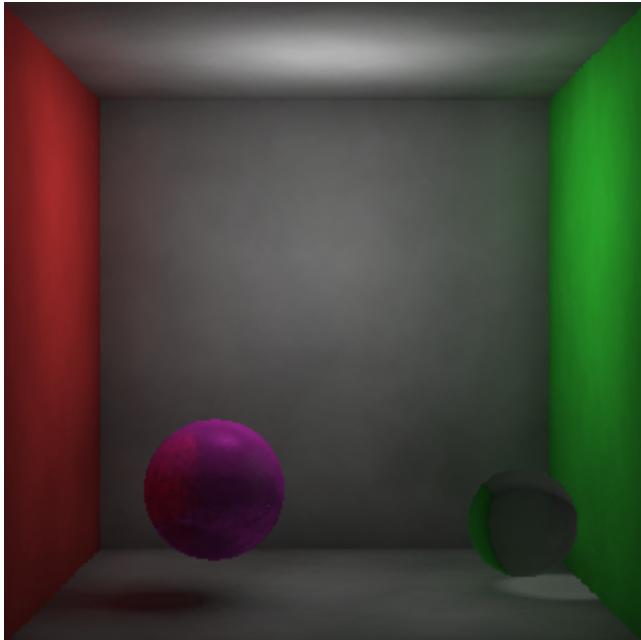
readObject :: Binary a => FilePath -> IO a
readObject = decodeFile
```

# Capítulo 4

## Resultados experimentales

A continuación se muestran algunos resultados obtenidos variando los kernels desarrollados que son los que se encargan de darle peso a los fotones. También se muestra una pequeña prueba visual resultante de variar entre inRadius y k Nearest.

Kernel box inRadius

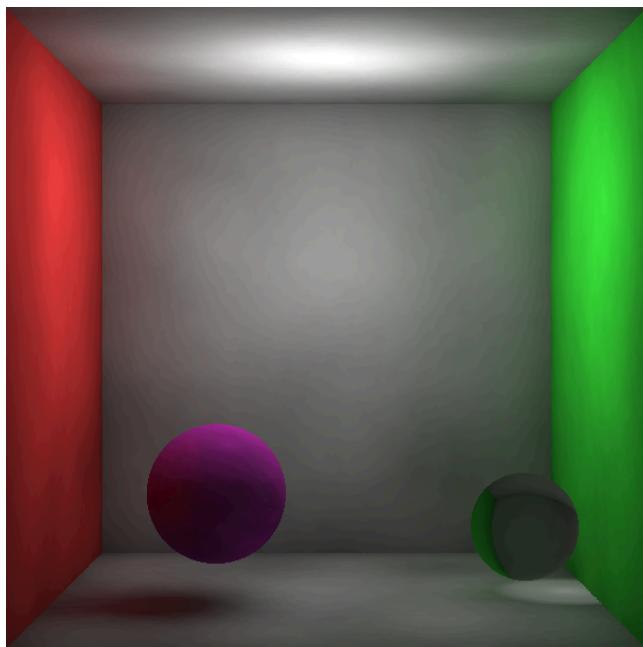


El primer y más sencillo kernel es un box kernel que se genera de esta manera

```
(1/(radio * radio * pi))
```

dándole un peso constante a todos los fotones generando una imagen bastante uniforme en todo momento.

### Kernel distancia lineal inRadius

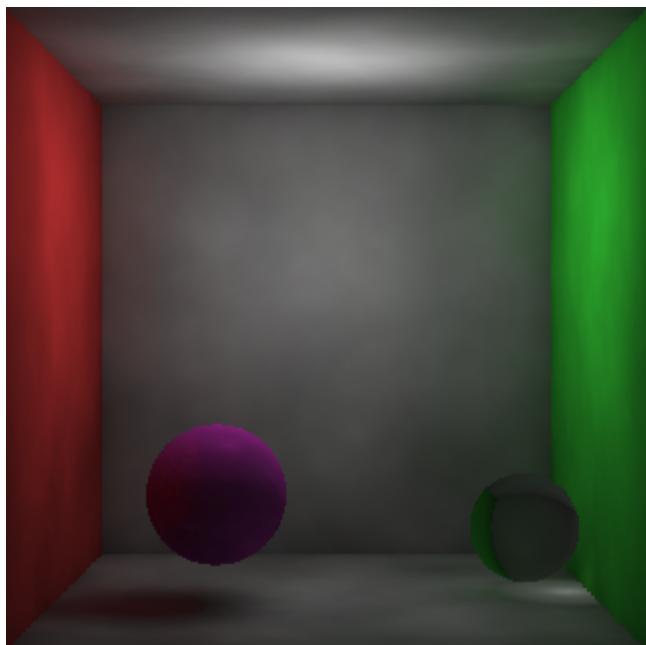


El siguiente Kernel que se ha medido utiliza la distancia del fotón al punto de colisión para darle distinto peso a los fotones.

```
((1 / (1 + distFot (colObj obj) photon)))
```

Con diferencias con el kernel anterior en esta encontramos más potencia de la luz, además de diferentes efectos, por ejemplo en la cáustica formada, donde más al centro de esta más potencia tiene, aunque tampoco es muy visible.

### Kernel distancia cuadrática

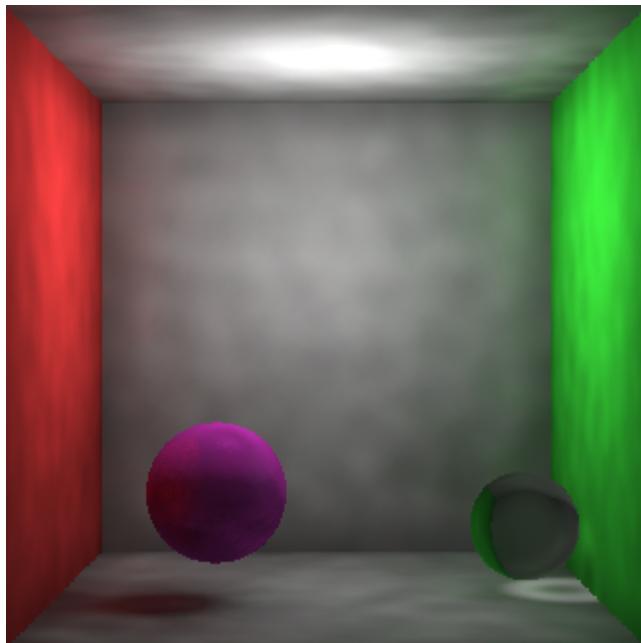


Junto con el anterior, se ha probado el mismo kernel pero cogiendo esta vez la distancia cuadrática.

```
((1 / (1 + distFot (colObj obj) photon))**2)
```

En esta podemos observar una bajada de nuevo en la potencia de la luz, ya que los fotones más alejados ahora tienen menos aportación que antes, sin embargo gracias a eso podemos notar una imagen con las luces más parejas que en el anterior y la cáustica se nota un poco más.

## Kernel Gaussiano inRadius



Finalmente, el último kernel desarrollado es un kernel Gaussiano donde tiene en cuenta la distribución de todos los fotones encontrados.

```
(fGaus photons obj photon)
```

Para desarrollar este kernel se han tenido que desarrollar funciones auxiliares que se dejan debajo que hacen referencia a las funciones Gaussianas.

De la imagen podemos notar de vuelta la subida de la luz y bastante más imperfecciones en las paredes techo y suelo de las sombras debida a la distribución de los fotones en la escena.

Además la cáustica también ha cambiado su forma debido a cómo se pesan los fotones.

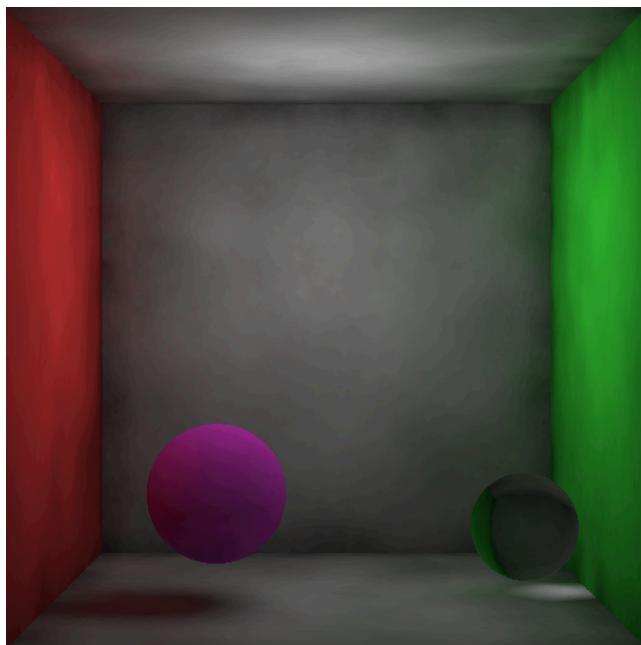
```
media :: [Float] -> Float
media xs = sum xs / fromIntegral (length xs)

varianza :: [Float] -> Float
varianza xs = sum (map (\x -> (x - m) ^ 2) xs) / fromIntegral (length xs)
  where m = media xs

desviacionEstandar :: [Float] -> Float
desviacionEstandar xs = sqrt (varianza xs)

fGaus :: [Foton] -> Obj -> Foton -> Float
fGaus photons obj fot = if isNaN result then 0 else result
  where
    !list = map (distFot (colObj obj)) photons
    a = 1 / (c * sqrt (2*pi))
    b = media list
    c = desviacionEstandar list
    x = distFot (colObj obj) fot
    result = a * exp (-(((x-b)**2) / (2*c**2)))
```

K Nearest 200 fotones, kernel distancia lineal



A su vez se ha querido observar las diferencias con cómo se vería la imagen cogiendo x Fotones dados. En esta imagen encontramos varias diferencias con respecto a su homóloga con inRadius. La primera y más notorias son las esquinas que están más oscurecidas debido a que coge los fotones más cercanos, sin embargo, como también se realiza un filtrado de los fotones donde solo tiene en cuenta los de su propio objeto, prácticamente no está teniendo en cuenta la mitad de los fotones medidos.

Además también se observa en la cáustica generada por la esfera de cristal que está más concentrada. Ya que ahí se encuentran muchos fotones en un radio pequeño, el K Nearest prácticamente solo tiene en cuenta los fotones de la cáustica incluso en su márgenes, sin embargo el inRadius, al solo importarle la distancia a los que tenía que mirar los fotones se podría agrandar más la cáustica.

# 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 Photon Mapper 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 Photon Mapper 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/>