Introducción a la Computación Gráfica

Obligatorio 2

Whitted Ray Tracing

Integrantes

Nombre	**CI**	**Correo**
Emiliano Botti	4.670.479-1	emiliano.botti@fing.edu.uy
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se tratan los aspectos generales del proceso de desarrollo del **Obligatorio 2**, utilizando la plantilla otorgada por el cuerpo docente como base estructural. A continuación, una breve descripción del trabajo que se detalla en cada sección:

1. En la presente sección, se aborda el problema como tal y el análisis realizado previo a la construcción de la solución.
2. En la sección **Desarrollo** 2, se detalla tanto la arquitectura de la solución, como los aspectos técnicos de desarrollo e implementación, poniendo especial énfasis en los puntos relacionados a los algoritmos implementados. En un segundo plano, se tratan los aspectos de implementación accesorios, que permiten generar los archivos intermedios y demás. Además, se agrega un historial de evolución de resultados a lo largo del desarrollo del obligatorio, así como distintas pruebas realizadas para medir la performance de la solución.
3. En la sección **Conclusiones** 3, se realiza un análisis posterior a la construcción de la solución, repasando conceptos aprendidos y puntos donde hubo dificultades.
4. Finalmente, en la sección **Trabajo futuro** 4, se mencionan todos los puntos que quedaron pendientes en el proceso de desarrollo y no fueron realizados por limitaciones de tiempo.

1.1. Definición del problema

El objetivo principal de la solución es generar una imagen a través del algoritmo *whitted ray tracing*, tomando como base una escena predeterminada que cumpla ciertos requisitos detallados en la consigna. A continuación se realiza un desglose de las funcionalidades básicas incluidas dentro de este alcance:

1. Implementación del algoritmo en cuestión, permitiendo el render de objetos de diversos tipos (incluyendo varias morfologías, tipos de material, etc.).
2. Implementación de un módulo para la carga dinámica de la escena, pudiendo determinar en un archivo XML los objetos de distinta índole que se cargaran, así como las características de cada uno (detalladas más adelante).
3. Implementación de un módulo para el guardado de la imagen generada, así como todos aquellos resultados intermedios que sea de interés acumular.

En las siguientes secciones se baja a tierra el desarrollo de cada requisito, no entrando ahora en detalle sobre su definición, ya que esta es tratada en la consigna del obligatorio.

1.2. Análisis del problema

El problema en cuestión incluye múltiples aristas, no obstante, el foco principal del mismo se encuentra mucho más centralizado en la implementación del algoritmo en cuestión. Aprovechando que C++ es un lenguaje con orientación a objetos, se determinó como directriz el diseñar un módulo para cada una de las tres actividades mencionadas en la sección anterior, con el objetivo de encapsular los distintos comportamientos.

Para representar la escena, se determinó hacerlo como una colección de objetos de distinto tipo, utilizando un esquema de herencia para atributos de carácter general como la posición de un objeto, y luego implementaciones más específicas para las dimensiones y otras características del estilo. En el proceso, separar la lógica del algoritmo de manera tal que distintos objetos cuenten con distintas responsabilidades dentro del proceso de generación de la imagen, tratando de minimizar el acoplamiento pero a su vez facilitando su entendimiento.

2. Desarrollo

2.1. Arquitectura

Teniendo en mente las directrices determinadas al momento de analizar el problema, se determinó la construcción de los siguientes módulos con sus respectivas responsabilidades:

- `Renderer` - El módulo principal, donde se instancian los múltiples requerimientos de la letra. El código que ejecuta puede ser dividido en la siguiente etapas:
  1. **Cargado de la escena:** Se lee el archivo de configuración `config.xml` y se generan los objetos correspondientes para la cámara, luces y los objetos de la escena en general.
  2. **Creación del historial:** Se crea una carpeta con fecha y hora actual para guardar el historial de imágenes generadas por la ejecución del algoritmo.
  3. **Generación de imagen principal:** Teniendo los objetos de la escena cargados, se invoca la llamada al algoritmo de *whitted ray tracing* para generar la imagen principal y guardarla en el directorio del historial. Se pasa parámetro `ImageIs::FullResult` para especificarle al algoritmo que genere la imagen con todos los componentes de iluminación y sombreado.
  4. **Generación de imágenes auxiliares:** Utilizando la misma escena, se vuelve a llamar al algoritmo pero variando el parámetro `ImageIs` para generar el conjunto de imágenes auxiliares, buscando aislar los resultados de aplicar cada uno de los coeficientes y colores por separado.
- `Managers` - Módulo donde se definen los manejadores principales de la solución, en base a diversas responsabilidades:
  - `Render` - Implementa el algoritmo de *ray tracing*, junto a todos los componentes que renderizan los diferentes resultados intermedios.
  - `Scene` - Encapsula la generación de la escena, cargando los distintos elementos desde un archivo XML (cámara, luces, objetos).
  - `Story` - Maneja el historial de los directorios fechados junto con los archivos de los diferentes resultados.

  Se destaca que todos los manejadores son de única instanciación, es decir, implementan el patrón de diseño *singleton* y por lo tanto son de acceso global, siempre y cuando el módulo incluya las dependencias necesarias.
- `Helpers` - Módulo donde se encapsulan diversos métodos de funcionalidad específica y utilizados en diversas áreas del programa, siendo los mismos:
  - `FileHelper` - Responsable de la creación de directorios (`create_directory`) y el guardado de imágenes (`save_image`).
  - `MathHelper` - Responsable de encapsular cálculos matemáticos a usar en el resto de la aplicación, como por ejemplo, operar con colores sumándolos (`add_colors`), mulitplicándolos entre sí (`multiply_colors`), o con un escalar (`scale_color`).
- `Models & Structures` - Módulos donde se definen representaciones de los objetos que intervienen en la implementación del algoritmo de *ray tracing*, categorizándose según su participación dentro del algoritmo, ya sea como *modelos* (aquellos que representan objetos dentro del espacio) o como *estructuras* (aquellos utilizados de manera auxiliar para representar conceptos más abstractos). Se expanden en la sección 2.2.3.
- `Constants` - Módulo de determinación de constantes, las cuales son de uso extendido por toda la aplicación y buscan sustituir un comportamiento configurable. Se expande su descripción en 2.2.5.
- `Resource Files` - Contiene los archivos XML con la información paramétrica de cada uno de los elementos a renderizarse en el escenario. Hay diferentes archivos que representan diferentes escenas, utilizándose de forma automática el archivo `config.xml`.

2.2. Implementación

2.2.1. Entorno de desarrollo

- **Sistema Operativo:** Windows 10 x64
- **Lenguaje:** C++11
- **IDE:** Visual Studio C++
- **Compilador (Release):** `/O2` (Maximum Optimization (Favor Speed)), `/Oi` (Enable Intrinsic Fucntions) `/GL` (Whole Program Optimization)

2.2.2. Bibliotecas utilizadas

Para este obligatorio la única biblioteca que se agregó a la base de C++11 fue **FreeImage**, empleándose la misma para la generación y escritura de imágenes en disco, utilizando matrices como estructura base.

Además de esto, se agregaron dos implementaciones de código de terceros de manera directa, siendo las mismas:

- **Vector3** [14]: Implementación de vectores y operaciones vectoriales con los mismos. Se destaca como esta misma implementación fue utilizada en el obligatorio anterior, aunque en esta ocasión se modificó su nombre a **Vector**.
- **TinyXML-2** [15]: Implementación de parser sencillo para lectura y escritura de archivos XML. En esta ocasión sólo se utilizaron las funcionalidades relacionadas a la lectura.

Por último, se destaca que un subconjunto de las funcionalidades desarrolladas usaron como referencia de implementaciones de terceros. Dichas funcionalidades fueron adaptadas a la arquitectura y contexto de la solución, y en general sirvieron más para facilitar el entendimiento del algoritmo que como fuente de código. No obstante, resulta de interés mencionarlas:

- **C++ Path Tracing Rendering (Medium)** [3]: Consultado a nivel general para la estructura y arquitectura del algoritmo.
- **RayTracing (iceman201, Liguo Jiao, GitHub)** [7]: Consultado para revisar y validar el funcionamiento de la intersección con cilindros sin tapas.
- **Lambda (TomCrypto, Thomas Bénéteau, GitHub))** [12]: Consultado como referencia para la implementación de refracción.
- **Volume 1: Foundations of 3D Rendering (Scratchapixel)** [2] [5] [8] [9] [10]: Se accedieron a varios snippets de códigos relacionados.
- **Using FreeImage 3.11 for Image Output (Niels Joubert, CS184 TA)** [13]: Consultado para obtener lineamientos y un ejemplo de código para convertir una matriz a una imágen `FIBITMAP` y guardarla con **FreeImage**.

2.2.3. Estructuras de datos

A continuación, se detallan las estructuras de datos utilizadas en la solución. Se destaca que se separan en dos grupos según lo que representan:

Estructuras:

- `Color:`
  Estructura auxiliar compuesta por cuatro valores `double` en el rango $[0, 1]$, representando los colores en un esquema `RGBA`. Se destaca que en determinados escenarios de la ejecución del algoritmo, la cota superior puede pasarse para algún componente del color. No obstante, dicho fenómeno se controla y se detalla en la siguiente sección.
- `Vector:`
  Estructura auxiliar utilizada para representar coordenadas de un espacio vectorial de 3 dimensiones, implementando operaciones generales como producto escalar, vectorial, entre otros comportamientos de vectores en cuestión. Se utiliza extensivamente en el resto de la solución.
- `Polygon:`
  Estructura auxiliar utilizada para representar polígonos de 3 vértices (triángulos), siendo modelados como vectores del espacio. Se utiliza exclusivamente como parte del componente **Mesh**, mencionado más adelante.
- `Image:`
  Estructura auxiliar utilizada para representar matrices de color, es decir, imágenes. Cuenta con un método para convertir la matriz en cuestión en un objeto del formato requerido por *Freeimage* para almacenar imágenes.

Modelos:

- `Camera:`
  Representación de la cámara que observa la escena, modelando como vectores la posición para el ojo y otra para el centro de la ventana de imagen (para la cual, también se modelan las dimensiones de largo y ancho). Cada escena contiene una única cámara.
- `Light:`
  Representación de una luz focal, conteniendo una posición (modelada como vector9 y un color (modelado con la estructura color). Cada escena contiene una colección de luces, pudiendo tener tantas como se determine en el archivo de configuración de escena.
- `Ray:`
  Representación de vectores de proyección utilizados por el algoritmo, contando con un vector como posición de origen y un vector auxiliar para representar su dirección. Se utilizan para representar las proyecciones del ojo de la cámara hacia la ventana de la misma, así como para modelar los rayos de sombra proyectados desde cada objeto hacia las fuentes de luz focales. También representan los rayos producto de la reflexión y refracción. Son creados a demanda, a medida que el algoritmo se va ejecutando.
- `Object:`
  Representación de los objetos ubicados en la escena. Los mismos tienen distintas propiedades generales como posición (vector central), color, material, etc. y métodos abstractos como `intersect` y `get_normal`, siendo ambos de interés general para el algoritmo (el primero recibe un `ray` y determina la distancia del origen del mismo hasta el punto de contacto con el objeto, el segundo recibe un punto de contacto con el objeto y determina el vector normal a la superficie en dicho punto).
  Dentro de la implementación de `Object` se determinan dichas propiedades generales, pero al momento de instanciar cada objeto, se utiliza una clase hija que modele mejor el tipo de objeto en cuestión, también implementando el comportamiento de los métodos abstractos. Cada escena contiene una colección de objetos, pudiendo tener tantos como se determine en el archivo de configuración de escena. A continuación,

se detallan las propiedades específicas de cada tipo de objeto implementado:

- `Sphere`: Representación básica de una esfera, modelando únicamente su radio (ya que el resto de datos son compartidos con las otras implementaciones de objetos).
  Intersección: Tomando el rayo de posible contacto, se plantea la ecuación de la superficie de la esfera y a su vez se proyecta el rayo en cuestión, buscando una solución que satisfaga ambas ecuaciones. En caso de encontrarla, se computa la distancia del origen del rayo hacia el punto de contacto, devolviéndose como resultado del algoritmo.
  Normal: Tomando el punto de contacto, se traza un vector desde el centro de la esfera hasta dicho punto, devolviéndose el mismo como normal.

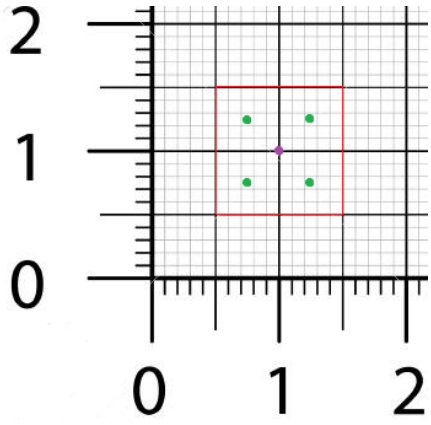
- `Cylinder`: Representación básica de un cilindro paralelo al eje vertical, modelando únicamente su radio y su altura (no incluyendo ángulo por falta de tiempo y complejidad a la hora de desarrollar el resto de funcionalidades).
  Intersección: Tomando el rayo de posible contacto, se sigue la misma idea que con la esfera: se proyecta el rayo y se plantea la ecuación de superficie del cilindro. Si se encuentran soluciones, se comprueba que estas se encuentren acotadas a la altura del cilindro en cuestión. En caso contrario, se comprueba que la intersección se pueda dar con el plano en el que se encuentre alguna de las dos bases. Si se encuentra una solución, se comprueba que la misma esté acotada al espacio interno de la circunferencia provista por la base del cilindro. En tal caso, se computa y devuelve la distancia como resultado del algoritmo.
  Normal: Tomando el punto de contacto, se controla primero que el mismo se encuentre en alguna de las bases del cilindro y en caso de ser así, se retorna un vector unitario paralelo al eje vertical y con la dirección correspondiente (arriba para base superior, abajo para base inferior). En caso de que el punto no se encuentre en ninguna de las dos bases, se asume que es parte del cuerpo del cilindro, y se sigue una lógica similar a la de la esfera, trazando un vector desde el eje central hacia el punto de contacto. No obstante, en este caso se toman en cuenta solo las coordenadas x y z, ya se proyecta el centro del cilindro como un eje.

- `Mesh`: Representación básica de un conjunto de triángulos, los cuales pueden compartir vértices o no. Se utiliza una colección de objetos de la clase `Polygon`, no controlando la adyacencia entre los mismos ni utilizando representaciones que optimicen la repetición de vértices (no implementado debido a falta de tiempo).
  Intersección: Tomando el rayo de posible contacto, se recorre la colección de polígonos que componen la malla en cuestión, comprobando para cada uno de ellos que la proyección del rayo intersecte con el plano en el que el polígono se encuentra. En caso de que esto suceda, se comprueba que el punto de contacto se encuentre delimitado dentro de las aristas del polígono y en caso afirmativo, se computa y retorna la distancia del origen del rayo hacia el punto de contacto en cuestión. Teniendo en cuenta que esto se realiza para cada polígono de la malla, se devuelve como distancia final la que se encuentre más cerca al origen del rayo, ya que la misma representa al punto de contacto de la cara más cercana al mismo, siendo físicamente la única cara con intersección posible.
  Normal: Tomando el punto de contacto y el rayo que lo proyecta, se recorre la colección de polígonos que componen la malla y se comprueba que el punto de contacto se encuentre delimitado por las aristas del mismo. En caso afirmativo, se computa el producto escalar entre la normal de dicho plano y la dirección del rayo. Si dicho producto es no positivo, la normal computada es la correcta y se retorna. En caso contrario, se retorna un vector opuesto a la normal computada, ya que implica que el rayo está chocando con la otra cara del polígono en cuestión.

2.2.4. Algoritmos implementados

A continuación, se específican los algoritmos implementados, mencionando el algoritmo principal de *whitted ray tracing*, así como detallando aquellos otros algoritmos que modifican o complementan el algoritmo principal en cuestión:

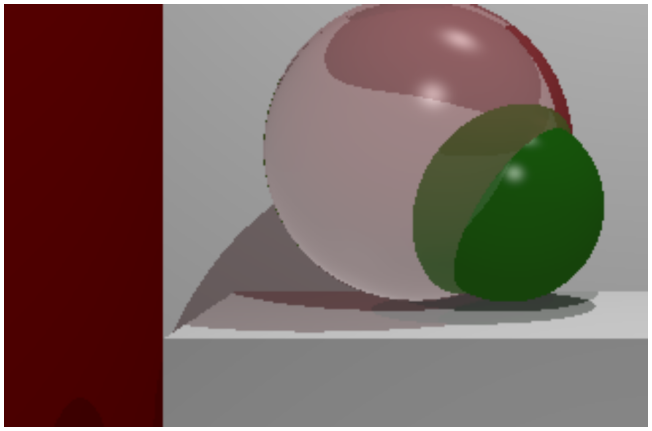
- **Whitted Ray Tracing:**
  Teniendo en cuenta que es el algoritmo principal y fue estudiado en clase, no se hace mucho hincapié en el mismo. No obstante, se considera importante mencionar las referencias teóricas utilizadas durante el proceso de desarrollo, con el objetivo de complementar lo visto en el curso:
  - **Recursive Ray Tracing (Techincal University of Budapest)** [1]: Base teórica del algoritmo Whitted Ray Tracing. Usada para complementar las clases y PPTs del curso.
  - **Ray Tracing (IRISA, Kadi Bouatouch)** [6]: Presentación del algoritmo de Whitted Ray Tracing y su aplicación a distintas formas geométricas. Esta referencia se consultó para complementar la base teórica en cuanto a las intersecciones con cilindros (para intentar resolver el problema que tenemos con los mismos).
  - **Reflection and Refraction (CSE Ohio 681)** [11]: Presentación del problema de la reflección y la refracción en el *whitted ray tracing*. Usada para complementar la base teórica acerca del tema.
  - **Calculate if vector intersects sphere (Math Stackexchange)** [4]: Base matemática para la intersección con esferas. Usada como referencia en la implementación de la misma.
- **Anti-aliasing:**
  Se implementó un anti-aliasing básico, con la política de que en el que en lugar de proyectar un sólo rayo por pixel, se proyectan otros cuatro rayos adicionales (variando su distancia horizontal y vertical en $\mp$ 0.25 píxeles respecto al pixel central) y el color resultante es el promedio de los colores de esos cinco rayos. En la primer imagen se muestra la distribución de puntos en el anti-aliasing (siendo el punto violeta el rayo sin usar anti-aliasing) y el punto violeta y los verdes usando anti-aliasing. En la segunda imagen se ve una imagen resultante a la que se le aplicó zoom para apreciar como los bordes de las sombras se ven más suaves luego de usar el algoritmo en cuestión.



Cuadro 1: Diagrama del Anti-aliasing

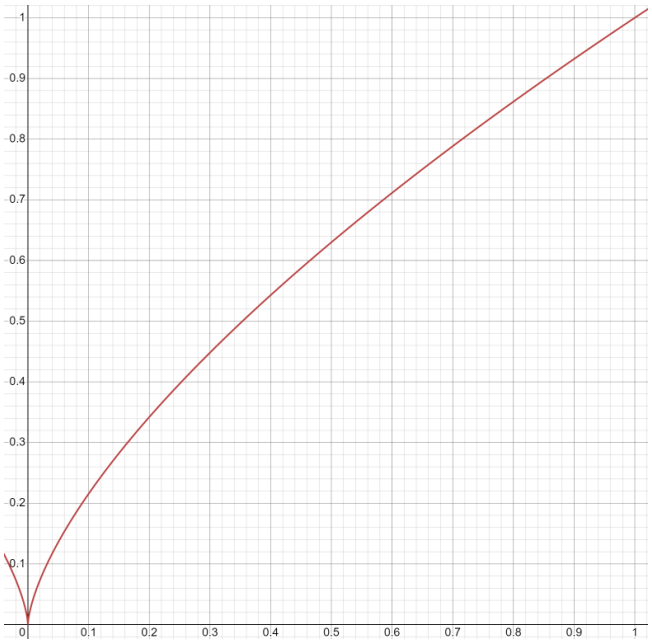
Cuadro 2: Resultado del anti-aliasing (zoom)

- **Sombras con color:**
  A modo de darle un poco más de realismo al algoritmo, se optó por implementar que, si un objeto se interpone entre un punto y una fuente de luz, y el mismo no es opaco (tiene un coeficiente de transmisión distinto de cero), aparte de proyectar una sombra (según los coeficientes de transmisión de los objetos entre la fuente de luz y el punto), el componente de la sumatoria afectado por dicha luz sea también multiplicado por el producto del color difuso del objeto intersectado y el coeficiente difuso del mismo. De esta manera, las sombras adquieren un poco del color del objeto que las proyecta.



Cuadro 3: Sombreado de color (zoom)

- **Gamma Correction:**
  Durante la etapa de normalización de los colores resultantes, se realiza una normalización min-max de los mismos (para traducir los coeficientes de cada color de vuelta a la escala 0..1), y a modo de balancear la intensidad de colores de los mismos (para contrarrestar el caso que un punto muy brillante oscureció mucho el resto de colores) se aplica una corrección gamma, elevando cada elemento del color por el coeficiente configurado (0,667 por defecto).



Cuadro 4: Gráfica de color para gamma correction

2.2.5. Archivo de escena

Para la carga dinámica de una escena en cuestión, se utilizaron archivos XML con un formato en particular. A continuación, se detalla un esquema del formato general que debe seguir un archivo `config.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<scene>
    <camera>...</camera>
    <lights>...</lights>
    <objects>...</objects>
</scene>
```

La cámara es representada por un único objeto `<camera>`, mientras que `<lights>` y `<objects>` son colecciones de objetos `<light>` y `<object>` respectivamente, pudiendo tener uno o más de cada uno.

El objeto `<camera>` cuenta con las siguientes propiedades:

- **Posición:** (`<position>`) Establece la posición con las coordenadas del espacio tridimensional `<x>`, `<y>` y `<z>`, marcando la posición en el espacio del ojo de la cámara.
- **Posición de la ventana:** (`<window_position>`) Establece la posición con las coordenadas del espacio tridimensional `<x>`, `<y>` y `<z>`, marcando la posición del centro de la ventana de la cámara.

Los objetos `<light>` cuentan con las siguientes propiedades:

- **Posición:** (`<position>`) Establece la posición con las coordenadas del espacio tridimensional `<x>`, `<y>` y `<z>`.
- **Color:** (`<color>`) Establece el color con la cuaterna RGBA `<r>` (red/rojo), `<g>` (green/verde), `<b>` (blue/azul) y `<a>` (alpha).

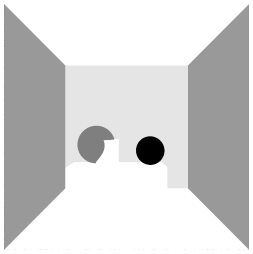
Los objetos `<object>` cuentan con las siguientes propiedades:

- **Posición:** (`<position>`) Establece la posición con las coordenadas del espacio tridimensional `<x>`, `<y>` y `<z>`.
- **Color difuso:** (`<diffuse_color>`) Establece el color difuso con la cuaterna RGBA `<r>` (red/rojo), `<g>` (green/verde), `<b>` (blue/azul) y `<a>` (alpha).
- **Color especular:** (`<specular_color>`) Establece el color especular con la cuaterna RGBA `<r>` (red/rojo), `<g>` (green/verde), `<b>` (blue/azul) y `<a>` (alpha).
- **Propiedades:** (`<properties>`) Conjunto de propiedades de cada objeto, incluyendo propiedades genéricas que todo tipo de objeto tiene (como el tipo de objeto y los coeficientes de material) y también propiedades específicas según el objeto en cuestión. A nivel general, las propiedades son:
  - **Tipo de objeto:** (`<type>`) Establece el tipo de objeto geométrico. Entre ellos: (`cylinder`) para el cilindro, (`sphere`) para la esfera, y (`mesh`) para la generación de objetos a partir de vértices triángulos.
  - **Coeficientes:** `<refraction_coef>`, `<transmission_coef>`, `<specular_coef>`, `<diffuse_coef>`, `<ambience_coef>` valores punto flotante que establecen los coeficientes del objeto.
  - **Reflectivo:** `<reflective>` Booleano `true/false` que establece si el objeto es reflejante o no.
  - **Otras propiedades:** (`<radius>`, `<height>`, `<polygons>`) Dependiendo del tipo de objeto, se pueden encontrar propiedades que determinan el radio del objeto (para esferas y cilindros), su altura (para cilindros) y una lista de polígonos (para las mallas). Se destaca que la lista de polígonos sigue cuenta con una cantidad variable de etiquetas `<polygon>`, donde cada uno cuenta con tres etiquetas `<vertex>` para modelar sus vértices. Cada uno de ellos cuenta con coordenadas `<x>`, `<y>` y `<z>` para modelar su posición

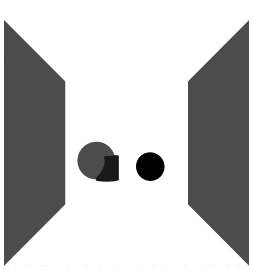
2.2.6. Imágenes auxiliares

Con el objetivo de facilitar el proceso de depuración, así como con la idea de poder observar resultados intermedios para la ejecución del algoritmo, se planteó la posibilidad de generar imágenes auxiliares que cuenten cuenten con un único componente de los varios que podría implementar. De esta manera, se generan dos tipos de imágenes auxiliares según el concepto:

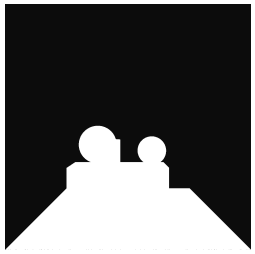
- **Coeficientes:** Imágenes en escale de grises que muestran para todo objeto intersectado el valor del respectivo coeficiente de cada imagen.
- **Colores:** Imagenes RGBA que muestran por separado el efecto que tiene la luz de ambiente y las luces focales tanto en el componente difuso como en el especular. La suma de colores en las mismas (posteriormente normalizada) daría como resultado una imagen con estos tres componentes (la imagen resultante del algoritmo excluyendo los componentes de reflexión y refracción).



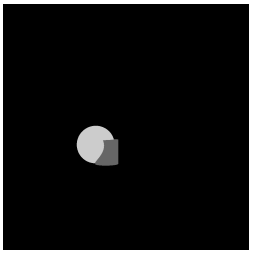
Cuadro 5: Coeficiente de Ambiente



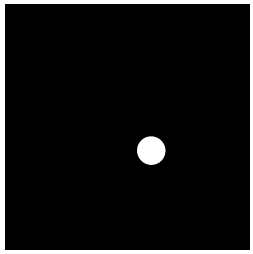
Cuadro 6: Coeficiente Difuso



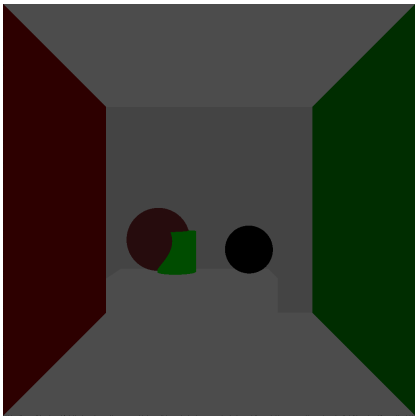
Cuadro 7: Coeficiente Especular



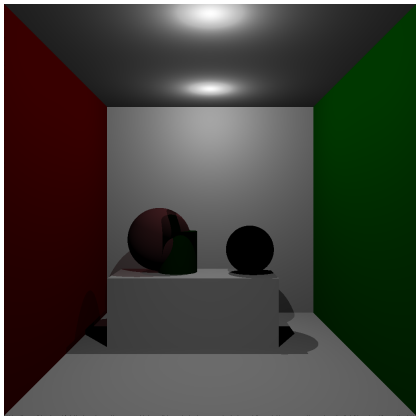
Cuadro 8: Coeficiente de Transmisión



Cuadro 9: Coeficiente Refracción (IOR)



Cuadro 10: Color de Ambiente



Cuadro 11: Color Difuso

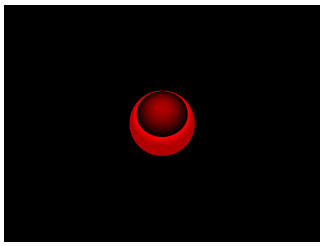
Cuadro 12: Color Especular

2.3. Obligatorio

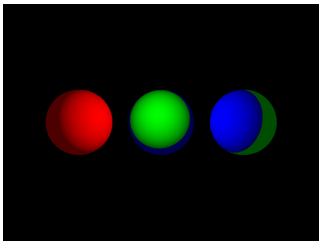
2.3.1. Historial y evolución de la aplicación

Junto a la entrega, se adjunta el historial de imágenes generado durante el proceso de construcción del algoritmo dentro de la carpeta *results*, en donde cada subdirectorio tiene como nombre la fecha y hora en la que fue generada, y dentro del mismo se encuentran las imágenes resultantes. También se adjunta una carpeta *highlighted_results* que contiene un subconjunto de las imágenes anteriores. En el nombre de cada una está reflejado la fecha y hora de cuando se obtuvieron.

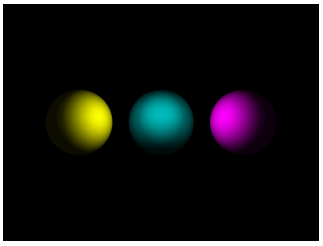
Se destacan ocho resultados intermedios, con el objetivo de mostrar superficialmente algunos de los desafíos a los que hubo que enfrentarse durante el proceso de desarrollo:



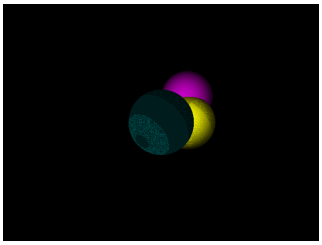
Cuadro 13: Introducción color Difuso. Los colores no normalizados hacían modulo 255



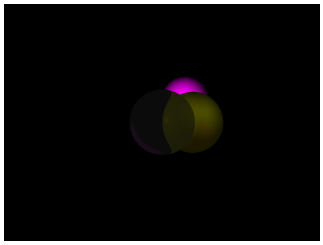
Cuadro 14: Problema con los colores (RBG en lugar de RGB)



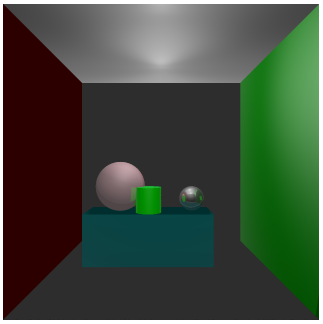
Cuadro 15: Implementación de Gamma Correction



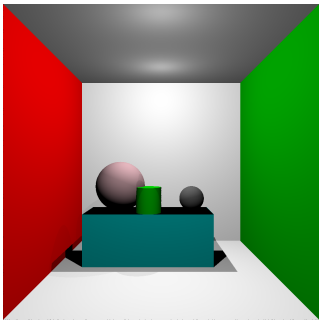
Cuadro 16: Problema de puntos con la refracción



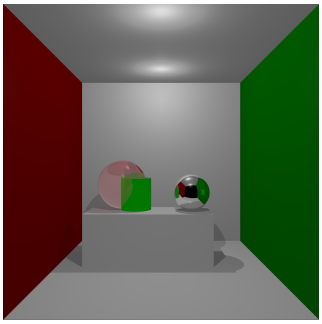
Cuadro 17: Refracción



Cuadro 18: Luz especular y escena base. Problema con normal de la malla de triángulos



Cuadro 19: Corrección de rayos de sombra. Problema con la normal de la mesa



Cuadro 20: Reflexión, atenuación con la luz y anti-aliasing

2.3.2. Estudio de performance

La evaluación experimental se realizó sobre el entorno de desarrollo 2.2.1, en un equipo que posee las siguientes caracteristicas:

- **Procesador:** Intel Core i5-4210U
- **RAM:** 8 GB DDR3
- **Disco:** SSD 480 GB

Respecto a la misma, se destacan los siguientes aspectos generales:

- Los tiempos registrados al realizar distintas ejecuciones de un mismo programa no fueron siempre constantes, sin embargo su variación osciló con un radio lo suficientemente bajo como para que las observaciones y conclusiones presentadas se preserven.
- El único tiempo que fue medido fue el de la función de *ray tracing* principal, esto es, excluyendo la escritura de imágenes y generación de imágenes auxiliares.
- La experimentación se dividió en dos áreas, la primera trabajando con la escena principal (consigna del obligatorio), en la cual se experimentó cuanto varían los tiempos del algoritmo acorde al anti-aliasing y a la profundidad máxima. La segunda no emplea el anti-aliasing y fija la profundidad en 6, mostrando los tiempos del algoritmo en diversas escenas distintas. A continuación se presentan estas escenas:

A continuación se adjuntan los resultados obtenidos para las distintas configuraciones paramétricas, tanto para la escena principal como para las variaciones de objetos:

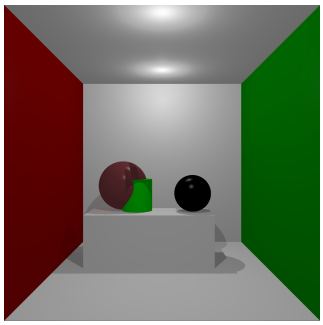
Antialiasing	**Profundidad**	**Tiempo** (s)
Sí	1	126.672
Sí	2	80.358
Sí	3	130.746
Sí	4	130.762
Sí	5	123.037
Sí	6	144.383
No	1	16.973
No	2	21.036
No	3	17.240
No	4	24.051
No	5	24.506
No	6	24.088

Cuadro 21: Evaluación de tiempos en escena principal

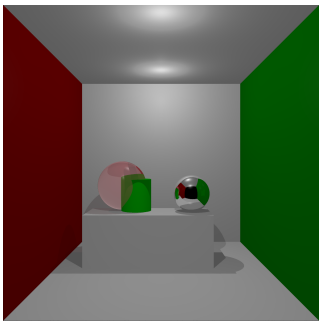
Escena	**Tiempo** (s)
Principal	24.088
Refracción con esferas anidadas (prof=6)	27.491
Refracción con esferas anidadas (prof=20)	27.184
Reflexión con un espejo	25.388
Reflexión con múltiples espejos	54.595
N Objetos	196.866
N Objetos difusos	120.476
3 Objetos difusas	1.218

Cuadro 22: Evaluación de tiempos en escenas de evaluación (sin antialiasing, profundidad 6)

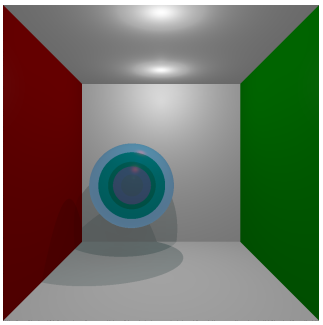
Por último, se adjuntan los resultados de distintas ejecuciones para diversas pruebas variando cantidad de objetos, naturaleza de los mismos, entre otros aspectos:



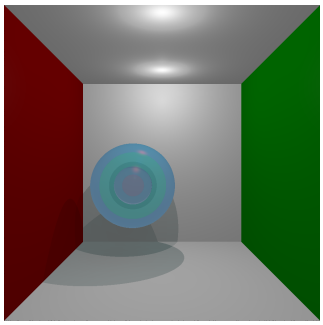
Cuadro 23: Escena principal c/antialiasing profundidad=1



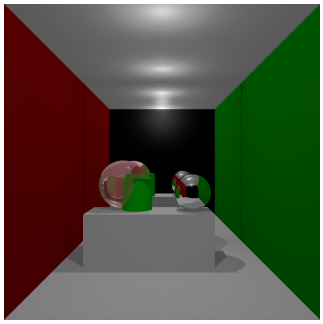
Cuadro 24: Escena principal c/antialiasing profundidad=6



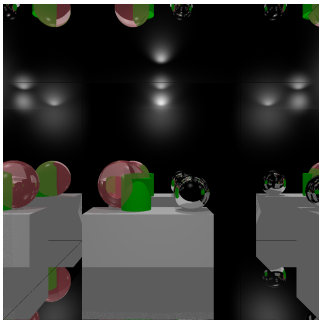
Cuadro 25: Refracción con esferas anidadas (prof=6)



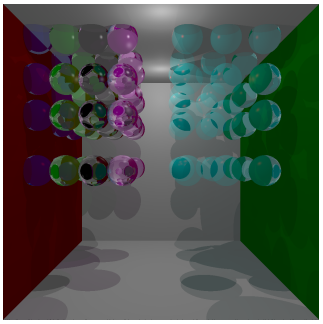
Cuadro 26: Refracción con esferas anidadas (prof=20)



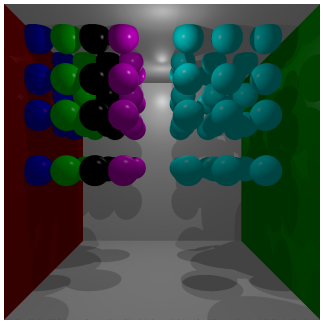
Cuadro 27: Reflexión con un espejo



Cuadro 28: Reflexión con múltiples espejos



Cuadro 29: N Objetos



Cuadro 30: N Objetos difusos

3. Conclusiones

Respecto a los tiempos de ejecución registrados se aprecia lo siguiente:

- Si bien el anti-aliasing mejora significativamente la calidad de imagen también implica un coste significativamente alto en el tiempo de ejecución (lanzando cinco veces más rayos de luz que el algoritmo sin anti-aliasing) y no aplicando ningún tipo de paralelización entre los mismos.
- La profundidad máxima no parece tener un coste muy significativo en el tiempo de ejecución, al menos para las escenas que probamos, de igual manera se aprecia como se obtuvieron tiempos más altos de ejecución.
- Se observó que al aumentar la complejidad de la escena tanto en cantidad y disposición de objetos como en características de materiales, el tiempo de ejecución aumentó notoriamente. Se concluye que no es una implementación apta para generar imágenes en tiempo real, aunque queda pendiente su prueba de performance en caso de implementar las múltiples técnicas de aceleración posibles.
- Por último se aprecia que algunos de los tiempos capturados hacen ruido, como por ejemplo el caso de la escena principal con anti-aliasing y profundidad 2 que demoró 80 segundos. Se asocia este problema a que, al correr el código en Windows, muchas veces el sistema operativo disminuía la prioridad del proceso durante su tiempo de ejecución. Por cuestiones de tiempo no se pudo re-capturar resultados ni intentar tomar alguna otra acción respecto como correr la experimentación sobre Linux.

A modo de conclusión general, se destacan los siguientes puntos:

- La construcción de un algoritmo de *ray tracing* para una escena con cantidad y variación de dinamismo entre los objetos resultó una experiencia enriquecedora desde múltiples aristas, principalmente cuando se tiene en cuenta la complejidad de los fenómenos físicos y la dificultad de llevar a código una implementación que capture sus peculiaridades.
- Si bien se considera que se llegaron a los requisitos mínimos, las dificultades asociadas a la complejidad del algoritmo no sólo entorpecieron el proceso de desarrollo, sino que afectaron a los resultados finales, arrastrando algunos errores de índole secundario.
- Luego de haber realizado el proceso de desarrollo, se identificaron múltiples cambios a nivel de arquitectura y acoplación de componentes que hubiesen facilitado tanto el desarrollo como la depuración del código generado. Por motivos de tiempo no se implementaron dichos cambios, pero el aprendizaje se mantiene para instancias futuras.

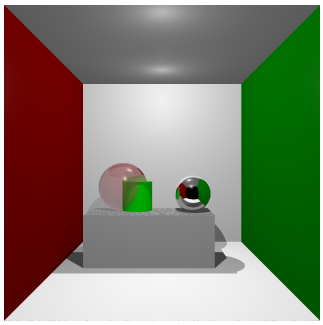
4. Trabajo futuro

A continuación se brinda una lista de funcionalidades que fueron contempladas y no pudieron realizarse ya sea por falta de tiempo o por fallas durante el proceso de desarrollo. Se pueden separar dichos aspectos en dos grupos:

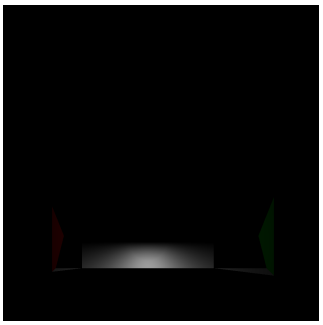
Errores:

- A la hora de determinar la normal de la malla de triángulos, existen escenarios donde se detecta que el punto solicitado corresponde a una cara incorrecta. Esto también ocurría en la intersección, pero se arregló retornando la distancia más cercana. Con la normal, el arreglo no resultó trivial. Se adjunta una imagen de la mesa en la que si su tapa se declara al final del XML hay problemas en su normal.
- A la hora de determinar la intersección con la malla de triángulos desde ángulos que vengan en direcciones distintas a la cámara (como de sombra o reflexión), se encontró que para estos rayos la mesa tiene un agujero en el costado. Se cree que este error está relacionado al anterior.

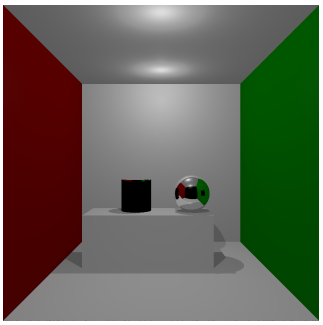
- Se observa en la esfera un reflejo del cilindro en la pared derecha, los rayos con ángulos como reflexión y rayos de sombra parecen estar colisionando con el cilindro en algún punto (erróneamente)
- Se aprecia una sombra del cilindro proyectada sobre la espera como si recibiera luz desde abajo. Se cree este bug está relacionado al anterior.
- Investigando los errores con el cilindro se probó hacerlo reflectivo para hallar que los rayos que proyectan al reflejar van en direcciones incorrectas, se sospecha un problema con la intersección o con la normal.



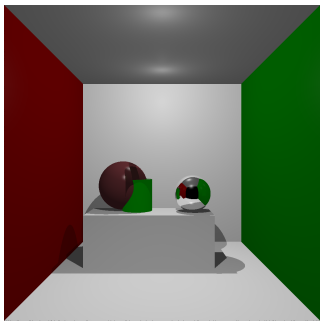
Cuadro 31: Error de la normal de la mesa cuando su tapa está última en el XML



Cuadro 32: Errores en los costados de la mesa intersectándo con los rayos de sombra de una luz dentro de la mesa



Cuadro 33: Cilindro reflectivo. Error en el reflejo de la esfera



Cuadro 34: Sombra del cilindro proyectada en la esfera refractante

Funcionalidades opcionales:

- Dar soporte a la definición de cilindros no verticales (con distintos ángulos).
- Aplicar técnicas de aceleración, como colisiones con bounding boxes al intersectar con mallas de polígonos.
- Paralelizar el algoritmo a modo que pueda beneficiarse de múltiples procesos e hilos (distribuyéndoles píxeles a computar entre los mismos).
- Factorizar la distancia en los componentes reflectivos y refractivos, aplicando atenuación en el color recibido de estos.
- Permitir el texturizado de objetos.
- Permitir la representación solidos avanzados (y agregar modelos 3D).
- Visualizar la generación de la imagen durante el tiempo de ejecución empleando SDL.
- Agregar aún más imágenes auxiliares variando profundidad, mostrando solo el impacto de la refracción y/o reflexión por separado.

5. Referencias

[1] Technical University of Budapest *Chapter 9: Recursive Ray Tracing*
http://www.fsz.bme.hu/~szirmay/ray.pdf

[2] Scratchapixel *An Overview of the Ray-Tracing Rendering Technique*
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted

[3] Medium *C++ Path Tracing Rendering*
https://medium.com/nicos-softwaredev-projects/path-tracer-project-75d734e2789

[4] Math Stackexchage *Calculate if vector intersects sphere*
https://math.stackexchange.com/questions/1939423/calculate-if-vector-intersects-sphere

[5] Scratchapixel *Ray-Triangle Intersection: Geometric Solution*
https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution

[6] IRISA, Kadi Bouatouch *Ray Tracing (Cylinder Intersection)*
http://www.irisa.fr/prive/kadi/Master_Recherche/cours_CTR/RayTracing.pdf

[7] iceman201 (Liguo Jiao, GitHub) *Cylinder*
https://github.com/iceman201/RayTracing/blob/master/Ray%20tracing/Cylinder.cpp

[8] Scratchapixel *Ray-Plane and Ray-Disk Intersection*
https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection

[9] Scratchapixel *Adding Reflection and Refraction, Introduction to Ray Tracing: a Simple Method for Creating 3D Images*
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/adding-reflection-and-refraction

[10] Scratchapixel *Reflection, Refraction and Fresnel, Introduction to Shading*
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel

[11] The Ohio State University College of Engineering, Computer Science and Engineering (CSE 681) *Reflection and Refraction*
http://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/reflection_refraction.pdf

[12] TomCrypto (Thomas Bénéteau, GitHub) *Refraction Smooth Glass Material*
https://github.com/TomCrypto/Lambda/blob/master/src/materials/smoothglass.cpp

[13] Niels Joubert, Stanford CS184 TA *Using FreeImage 3.11 for Image Output*
http://graphics.stanford.edu/courses/cs148-10-summer/docs/UsingFreeImage.pdf

[14] Lighthouse3d *View Frustum Culling*
http://www.lighthouse3d.com/tutorials/view-frustum-culling/

[15] TinyXML-2 *C++ XML parser*
https://github.com/leethomason/tinyxml2