UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PROGRAMACIÓN DE UN RAYTRACER, TEORÍA Y PRÁCTICA

TESIS

Que para obtener el título de

INGENIERO EN COMPUTACIÓN

P R E S E N T A N

CASTILLO GUERRA DANIEL

ESTRADA DOMÍNGUEZ GUILLERMO

GARCÍA CRUZ RENE

LEDEZMA MOLINA LUIS

LÓPEZ ROLDÁN EDGAR JESÚS

DIRECTOR DE TESIS

M.I. JUAN CARLOS ROA BEIZA

Ciudad Universitaria, Cd. Mx., 2017

Agradecimientos

A mi madre Maribel p*or haberme apoyado en todo momento, por sus consejos, sus valores, por la motivación constante que me ha permitido ser una persona de bien, pero más que nada, por su amor.*

Le doy gracias a mi padre Daniel por apoyarme en todo momento, por los valores que me ha inculcado, y por haberme dado la oportunidad de tener una excelente educación en el transcurso de mi vida. Sobre todo por ser un excelente ejemplo de vida a seguir.

A mis hermanos por ser parte importante de mi vida y representar la unidad familiar.

Le agradezco a Dios por haberme acompañado y guiado a lo largo de mi carrera, por ser mi fortaleza en los momentos de debilidad y por brindarme una vida llena de aprendizajes, experiencias y sobre todo felicidad.

A mis familiares y amigos

Esto es dedicado a quienes dieron mucho interés en que pudiera cumplir mis sueños y me ayudaron aunque para eso tuvieron que arriesgarse un poco, esto se lo dedico a mis abuelos que supieron aguantarme y reprenderme cuando era debido.

A mis tíos, primos y amigos que siempre estuvieron apoyándome en todo momento, siempre cuidándome de todo mal y a su vez enseñándome buenas costumbres de su época, y no solo eso también la disciplina que aprendí con ellos me ha servido tanto en la sociedad para comportarme.

A mis compañeros y Director de Tesis del PAT.

Al M. I. Juan Carlos Roa Beiza por su gran apoyo y motivación para la culminación de nuestros estudios profesionales y para la elaboración de esta tesis, a Guillermo Estrada Domínguez, Rene García Cruz, Luis Ledezma Molina y Edgar Jesús López Roldán por su apoyo ofrecido en este trabajo por su tiempo compartido y por impulsar el desarrollo de nuestra formación profesional.

¡Gracias a ustedes!

Daniel Castillo Guerra

A Dios, porque me dio la fortaleza necesaria para concluir el día de hoy este proyecto, por ser mi guía todos estos años de formación y desarrollo, por los dones que me ha entregado y por poner a lo largo de mi vida a todas y cada una de las personas que me trajeron hasta aquí.

A mis padres Margarita y Guillermo, por su apoyo y amor incondicional, por mi educación y los valores que inculcaron en mí, por su esfuerzo y trabajo constante, por haberme dado las herramientas y el cariño que me hacen ser el hombre que soy hoy, no tengo ni tendré como pagarles su enorme dedicación, empeño y sobre todo el derroche de amor que me han dado todos estos años.

A mi hermana Lilián, por estar siempre presente en mi vida, mi mente y mi corazón, por su amor, su apoyo y su confianza a lo largo de todos estos años y los que faltan.

A Paola, por ser un constante recordatorio de la confianza en mí mismo, por caminar paso a paso a mi lado estos 6 años con una sonrisa fiel, una mano firme y un corazón enorme que me levantan cada que caigo. Por soñar conmigo, sufrir conmigo, reír conmigo, planear conmigo y por sobre todo amar conmigo. Porque la culminación de este proyecto es sólo el primer pasó en lo que nos espera.

A mi familia y amigos por ser mi roca, mi constante y mi guarida, por sus consejos y cariño, simplemente por estar ahí siempre que más lo necesito.

A la Universidad Nacional Autónoma de México, mi máxima casa de estudios, por mi formación profesional.

A mis compañeros Rene García Cruz, Daniel Castillo Guerra, Edgar Jesús López Roldán, Luis Ledezma Molina, y al M.I. Juan Carlos Roa Beiza por todo su apoyo y dedicación, sin los cuales esta tesis no hubiera sido posible.

Desde el fondo de mi corazón.

¡Gracias!

Guillermo Estrada Domínguez

A mis padres por sus enseñanzas incondicionales que me han permitido enfrentar las adversidades a lo largo de mi vida y he superado exitosamente. Ellos son y serán siempre los causantes de mis logros obtenidos y a quienes se los dedico siempre.

A mis hermanos y sobrinos por los momentos tan gratos que me han hecho crecer en todos los sentidos.

A Omar por ser mi compañero inseparable y alegrar mi existencia.

A mis amigos y familiares quienes han complementado mi formación personal, académica y profesional con pláticas y comentarios enfocados siempre a seguir hacia adelante.

A DIOS por bendecirme todos los días.

A mis colegas de tesis Guillermo Estrada Domínguez, Luis Ledezma Molina, Edgar Jesús López Roldán, Daniel Castillo Guerra y al Mtro. Juan Carlos Roa Beiza del PAT por el empeño y dedicación invertido a la realización de esta trabajo.

Rene García Cruz

Utilizaré este pequeño espacio y mi mal uso de las palabras para intentar expresar mi más profundo agradecimiento a todas aquellas personas que con su ayuda me han llevado hasta donde me encuentro ahora.

A mi mamá, que toda la vida me ha apoyado, cuidado y enseñado; siempre has estado mi lado sin importar donde me encontrara; gracias por toda la comprensión, paciencia y el apoyo; sin ti esto no sería posible.

A mi hermana, sé que por ti me he convertido en una mejor persona, creciendo siempre juntos; no desearía que fuera diferente.

A mi papá, que me crió y del cual aprendí lecciones hasta en el último momento en que estuvo conmigo e incluso después de eso.

A mi gran familia, por cuidar de mí, me dieron asilo y me enseñaron lo que significa tener a alguien que te apoye en todo y cualquier momento.

Abuela Chila, Tía Gloria, Tía Mónica, Tía Yoyis, Tío Ricardo, Tío Jorge, Tío Quique siempre han ayudado, especialmente en los momentos difíciles, gracias.

A mis madrinas, padrinos y diversos vecinos que se convirtieron en mis amigos, gracias por su apoyo y bendiciones.

A mis amigos, que me ayudaron dentro y fuera de la escuela, les deseo lo mejor y espero seguir viéndolos.

A mis profesores, que me han formado durante toda mi vida, me han guiado a donde me encuentro.

A mi alma mater UNAM, por la formación que he recibido y el apoyo en cada etapa.

A mis compañeros René García Cruz, Daniel Castillo Guerra, Edgar Jesús López Roldán, Guillermo Estrada Domínguez, y al M.I. Juan Carlos Roa Beiza, por hacer posible esta tesis, gracias por su apoyo.

A todos ellos, muchas gracias.

Luis Ledezma Molina

Gracias quiero dar a Dios, *laberinto de los efectos y las causas*.

A mis padres por todo: mi madre, fuente de lo bueno, mi padre por su inteligencia.

A mi hermano Rey por todo su apoyo.

A mi hermana Alma por su compañía.

A mi abuela por todo el cariño.

A Timoteo por la amistad y las enseñanzas.

A Swami por las conversaciones y la amistad.

A Jazmín por aparecer en los momentos justos.

A Karen por su sonrisa y muchas cosas más.

A los buenos profesores de la Facultad de Ingeniería.

A la Universidad Nacional Autónoma de México por darme muchas de las mejores cosas de mi vida.

A mi familia por los buenos momentos y también los otros.

A mis amigos por los momentos especiales.

A todas las personas que he conocido y que significaron tanto.

A mis compañeros de tesis: Rene García Cruz, Guillermo Estrada Domínguez, Luis Ledezma Molina, Daniel Castillo Guerra y al M.I Juan Carlos Roa Beiza por la dedicación para la realización de este trabajo.

Y a todos los que me acompañaron desde el principio en esta aventura, algunos continúan hasta ahora. *Gracias Totales*.

Edgar Jesús López Roldán

ÍNDICE

Contenido

1 CAPÍTULO I. ANTECEDENTES Y MARCO TEÓRICO

1.1 Introducción

Con la llegada de las computadoras de cuarta generación, llego la generación de gráficos en tercera dimensión, basados en la representación de vectores, en la cual la computadora no almacena información sobre puntos, líneas y curvas en un plano bidimensional, la computadora almacena la posición de puntos, líneas y caras las cuales pueden construir un polígono o forma geométrica en un espacio de tres dimensiones.

Por esta razón, los interesados en aprender y controlar el funcionamiento de un Raytracer (Trazador de rayos), deben tener conocimientos básicos pero sólidos, de geometría analítica principalmente, ya que en la generación de gráficos asistidos por computadora es la herramienta base para poder modelar los comportamientos de la naturaleza dentro de las imágenes digitales, es decir, cuando se modela la trayectoria que sigue un rayo de luz hacia algún objeto y la colisión que se genera con los elementos de la escena, lo que se estará representando serán vectores en dos y tres dimensiones, por lo tanto se utilizarán operaciones con vectores para calcular distancias y determinar si existe intersección entre dos o más vectores, incluso si existe más de una intersección en diferentes puntos de los mismos vectores, debido a la desviación de la trayectoria de alguno.

El desarrollo de la computación gráfica, se debió a la alternativa que se ofreció a los efectos especiales, más reales y fáciles de crear, a diferencia de las técnicas tradicionales de animación, fue creado para fines académicos, sin embargo, cuando las aplicaciones de la computación grafica se utilizaron para la edición de series de televisión, películas y juegos de video, demostraron gran utilidad y principalmente el beneficio económico que puede generar el invertir en este campo para las grandes empresas que cuenten con el capital e infraestructura tecnológica necesaria en el desarrollo de los gráficos asistidos por computadora.

En el año 1952 se creó OXO el primer videojuego de la historia resultado de una tesis en la universidad de Cambridge del estudiante Alexander Sandy Douglas, en la cual mostraba como se realizaba la interactividad entre computadoras y seres humanos, este juego podía tomar sus propias decisiones en función de los movimientos del jugador que

transmitía las ordenes por medio de un cable telefónico integrado a la computadora que ejecutaba el programa del juego de video.

Mientras Alexander Sandy Douglas experimentaba con dispositivos de entrada y salida, iniciaba una de las revoluciones más importantes a nivel mundial en la historia, los videojuegos hoy en día son el otro gran campo económico de la computación gráfica, el cual día con día avanza a grandes pasos, ya que el mercado exige que los gráficos de los videojuegos, cada día sean más realistas, lo cual es aún más difícil si se toma en cuenta que la generación de gráficos consumen muchos recursos del procesador, además hay que interactuar con las ordenes que el jugador manda a las consolas de video mediante un control sincronizado con esta, y todo debe ser en tiempo real.

El desarrollo de este proyecto de tesis tiene como objetivo que los lectores conozcan el funcionamiento, desarrollo y la implementación de un Raytracer, para que adquieran las habilidades y conocimientos necesarios para el desarrollo de gráficos asistidos por computadora, herramientas que en la actualidad son muy utilizadas en diversos campos como: cine, publicidad e investigación científica principalmente, los cuales generan ganancias económicas muy grandes, esto se logrará mediante la elaboración de un curso estructurado de una serie de prácticas que guiarán paso a paso al lector en la generación de imágenes de escenas con objetos en tercera dimensión (3D), tomando en cuenta el comportamiento de los rayos que fungen como fuentes de luz, brindando la posibilidad de representar características físicas como transparencia, reflexión y refracción, mediante la interpretación de colisiones de los objetos con nuestras fuentes de luz.

Los alumnos que concluyan este curso tendrán conocimientos sólidos y comprenderán la importancia de la técnica del Raytracer, podrán generar imágenes con la cantidad de objetos que se quiera representar y observar el comportamiento de la luz dentro de la escena, lo que dará una gran sensación de realismo a la escena, obtendrá las bases de las técnicas que se implementan en la actualidad para la maquila de películas por grandes compañías como Pixar, quienes crearon la primera totalmente hecha con gráficos asistidos por computadora.

Debido a que este campo de las ciencias de la computación tiene gran importancia para distintas áreas del mercado comercial a nivel mundial, y la poca difusión que se le da a

este campo de la computación, se decidió elegir el Raytracer para este proyecto de tesis, porque se consideró que es de suma importancia contar con materiales bibliográficos en mayor número, que apoyen en la difusión, enseñanza y aprendizaje de estas técnicas que se hacen más importantes con el pasar de los años y que en países como el nuestro no cuente con material de referencia de fácil adquisición o comprensión, debido en mayor razón a que los libros que se tienen en la actualidad están escritos en un idioma diferente al español, regularmente inglés, lo que limita aún más la cobertura de personas interesadas en aprender computación gráfica.

En las primeras páginas de esta tesis se describirán los elementos base y definiciones principales de la computación gráfica, para lograr entender cómo funcionan y como se aplican sus principios fundamentales, que son la base para poder entender las técnicas del desarrollo de graficas asistidas por computadora, para después entrar más a detalle en la técnica, que para este proyecto será la más importante el Raytracer, se describirán sus conceptos y los que son más específicos, se detallará como es que se realiza una imagen mediante el trazado de rayos, que se interpretarán como fuentes de luz, las cuales definirán el color que ilumina cada elemento de nuestra imagen.

Después se presentarán cinco prácticas, en las cuales se desarrolla cada elemento de nuestra imagen, es decir, se aprenderá a representar los objetos mediante formas geométricas simples como puntos, rectas y planos, a los cuales se les llama primitivas, debido a que son los elementos primarios, ya que con ellos se representa cualquier objeto en la imagen y que además serán los límites para definir de que color se quiere iluminar cierta área de la imagen, mediante el trazado de rayos que serán la fuente de luz, muy parecido al efecto que produce el sol con el planeta y las colisiones o choques con diversas áreas definidas (los objetos), que ayudará a la vista a distinguir diferentes objetos en la escena, para concluir con un proyecto con el que se generarán imágenes con la cantidad de objetos y colores que se desean que se iluminen.

Dentro del contenido del capítulo tres, una vez que se conozca como elaborar una imagen, se implementarán efectos físicos a los objetos, lo que también influenciará en el color que se iluminarán los objetos en la escena, es decir, cómo se contemplan componentes que se manifiestan en la realidad, las imágenes serán muy realistas ya que

tendrán efectos de brillo, opacidad, reflexión y refracción, lo que provocará que las imágenes se vean en tercera dimensión, debido al realismo que generará para la vista.

Al final del proyecto de tesis, se aplicarán técnicas o extensiones con las cuales se podrán generar diversos comportamientos, los cuales pueden ahorrar desarrollo de muchos objetos, es decir, se implementarán modelos de comportamiento a los objetos, por lo tanto, con un solo objetos que se programe, se podrá generar una imagen con muchos objetos similares.

Para iniciar, se explicará que la computación grafica es el campo de las ciencias de la computación donde se utiliza tanto software como hardware para la generación de imágenes digitales, sus principales aplicaciones son las animaciones en 3D usadas en videojuegos, películas, creación y edición de video, edición de efectos especiales, edición de imagen y modelado.

El diseño asistido por computadora básicamente es una colección de entidades y formas geométricas simples como lo son puntos, líneas y circunferencias, que se pueden manipular a través de una interfaz entre la computadora y el usuario, generalmente conocida como interfaz gráfica, mostrada en un dispositivo de salida como los monitores. Permite diseñar en dos o tres dimensiones mediante geometría analítica, esto es, puntos, líneas, volúmenes y superficies para obtener un modelo numérico de un objeto o un conjunto de ellos que se quiera representar mediante gráficos.

Cada entidad u objeto que se encuentra dentro de la escena, es una colección de datos que permiten manejar la información de manera lógica, cada elemento de la entidad representa propiedades como color, estilo de línea, nombre y definición geométrica.

1.2 Conceptos generales de la computación gráfica

Durante estas tesis y las prácticas siguientes se estará tratando con temas y conceptos de la materia de computación gráfica, estos conceptos resultan esenciales para poder explicar, realizar y completar el tema de Raytracer; es por ello que se requiere dar un repaso de los términos y conceptos de computación gráfica antes de encontrarlos en las prácticas, principalmente para aquellas personas que no los conocen o solo requieren de un repaso por ellos para poder recordarlos.

La mayoría de los libros en computación gráfica están escritos desde la perspectiva de un programador que intenta completar una implementación de algún algoritmo, ya sea un juego de computadora o un sistema de animación. En contraste, el enfoque que se dará es desde un aspecto académico y de enseñanza, explicando los conceptos a alguien que solo tiene conocimientos básicos. Dichas personas pueden ser alumnos de distintas áreas de la computación o grados de estudios, que buscan un mejor entendimiento de los sistemas que tal vez ya utilizan en clases y quieren repasar, alguien que intenta empezar con los temas de gráficos pero no tiene experiencia previa, o incluso una persona curiosa que quiere conocer más acerca de la materia de computación gráfica.
Raytracer requiere de conocimientos matemáticos, desde coordenadas geométricas elementales hasta cálculos multidimensionales, y empezaremos con:

1.2.1 Conjuntos

Un conjunto es una colección desordenada de objetos del mismo tipo, con una regla para determinar si el objeto es dado en el conjunto. Los diagramas de Venn, introducidos por el matemático John Venn, Inglés en 1881, son muy útiles para la comprensión de las relaciones entre conjuntos. Un ejemplo de esto se puede ver en la figura 1.2.1.1

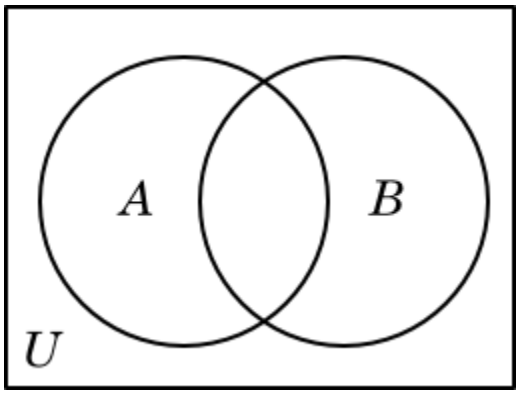


FIGURA 1.2.1.1 Diagrama de Venn de dos conjuntos A y B[1]

[1] http://math.cmu.edu/~bkell/21110-2010s/sets.html

Los objetos son un conjunto conocido como elementos. Hay dos maneras de describir o especificar los elementos de un conjunto. Una forma es mediante intensión, usando una regla o descripción semántica:

- A es el conjunto cuyos elementos son los primeros cuatro números enteros positivos.
- B es el conjunto de los colores de la bandera francesa.

La segunda forma es por extensión, es decir, una lista de cada elemento del conjunto. Una definición extensional se denota encerrando la lista de elementos entre llaves:

- C = {4, 2, 1, 3}
- D = {azul, blanco, rojo}

A menudo se tiene la opción de especificar un conjunto bien intencionalmente o extensionalmente. En los ejemplos anteriores, por ejemplo, A = C y B = D

Hay dos puntos importantes a tener en cuenta acerca de los conjuntos. En primer lugar, en una definición extensional, un elemento del conjunto se puede enumerar dos o más veces, por ejemplo, {11, 6, 6}. Sin embargo, por extensionalidad, dos definiciones de conjuntos que difieren sólo en que una de las definiciones de los conjuntos contienen varias veces un mismo elemento, de hecho, es el mismo conjunto. Por lo tanto, el conjunto {11, 6, 6} es exactamente idéntico al conjunto {11, 6}. El segundo punto importante es que el orden en que se enumeran los elementos de un conjunto es irrelevante (a diferencia de una secuencia o túpla). Podemos ilustrar estos dos puntos importantes con un ejemplo:

- {6, 11} = {11, 6} = {11, 6, 6, 11}

Aunque los objetos pueden ser de cualquier tipo, en un Raytracer, se usa un conjunto de números reales (punto flotante)

1.2.1.1 Subconjunto

Con frecuencia se utilizan conjuntos cuyos elementos pertenecen a un conjunto más amplio, que nos lleva al concepto de subconjuntos.

Un conjunto A es un subconjunto de un conjunto B si cada elemento de A también pertenece a B. Simbólicamente, se puede escribir como $A \subseteq B$

1.2.1.2 Par Ordenado

Un par ordenado (x, y) de los elementos es una secuencia de dos elementos en un orden definido. Un ejemplo común es un punto en el plano (x, y), donde siempre escribimos la coordenada “x” en primer lugar, seguido de la coordenada “y”.

Para dos conjuntos A y B, el conjunto de par ordenados de elementos (x,y) donde “x” pertenece a A y “y” pertenece a B es conocido como el producto cartesiano de los conjuntos A y B, y se denota como A x B.

1.2.2 Intervalos

Un intervalo es un subconjunto de la línea real que contiene al menos dos números, y contiene todo el número real que se encuentra entre dos de sus elementos.

Intervalo es un conjunto de números reales con la propiedad de que cualquier número que se encuentra entre dos números en el conjunto también está incluido en el conjunto. Por ejemplo, el conjunto de todos los números “x” que satisface $0 \leq x \leq 1$ es un intervalo que contiene 0 y 1, así como todos los números entre ellos. Otros ejemplos de intervalos son el conjunto de todos los números reales, el conjunto de todos los números reales negativos, y el conjunto vacío.

Los intervalos se definen asimismo en un conjunto totalmente ordenado arbitrariamente, tales como números enteros o números racionales.

El intervalo de números de entre a y b, incluyendo a y b, a menudo se denota [a, b]. Los dos números son llamados los extremos del intervalo.

1.2.3 Ángulos

En 2D las coordenadas (x, y), ángulos positivos se miden desde el eje “x” positivo, en un sentido antihorario, y los ángulos negativos se miden en sentido horario desde el eje “x” positivo, el ejemplo de la figura 1.2.3.1 muestra un ángulo a partir de dos vectores, que es como se usará en esta práctica.

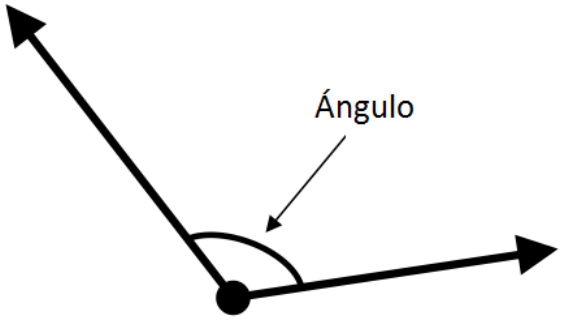


FIGURA 1.2.3.1 Ángulo[2]

Los ángulos se pueden especificar en grados y radianes. El radián es la unidad estándar de medida angular, que se utiliza en muchas áreas de las matemáticas. La longitud de un arco de una unidad círculo es numéricamente igual a la medida en radianes del ángulo que subtiende; un radián es un poco menos de 57.3 grados. Puesto que 2 pi radianes y 360 grados en un círculo,

- 1 radian = 180° / π = 57.29577…°.

1.2.4 Trigonometría

El teorema de Pitágoras para el triángulo de ángulo recto es $c^2=a^2+b^2$, si consideramos ese triángulo la función trigonométrica seno, coseno y tangente del ángulo A se define como:

- sen A = a / c,
- cos A = b / c,
- tan A = a / b = sen A / cos A

Las siguientes identidades están relacionados con el teorema de Pitágoras y se mantienen para cualquier valor:

- $sen^2 A+cos^2 B = 1$
- sen(A +/- B) = sen A cos B +/- cos A sen B
- cos(A +/- B) = cos A cos B -/+ sen A sen B

[2] http://www.webquest.hawaii.edu/kahihi/mathdictionary/A/angle.php

Cuando B = π/2, las ecuaciones anteriores se convierten en

- sen(A +/- π/2) = ±cos A
- cos(A +/- π/2) = -/+ cos A

1.2.5 Sistemas Coordenados

Un sistema de coordenadas es un sistema que utiliza uno o más números o coordenadas, para determinar de forma única la posición de un punto u otro elemento geométrico en un colector tal como el espacio euclidiano. El orden de las coordenadas es significativa, y siempre se identifican por su posición en una túpla ordenada y a veces por una letra, como en la coordenada “x", este se puede ver en la figura 1.2.5.1.

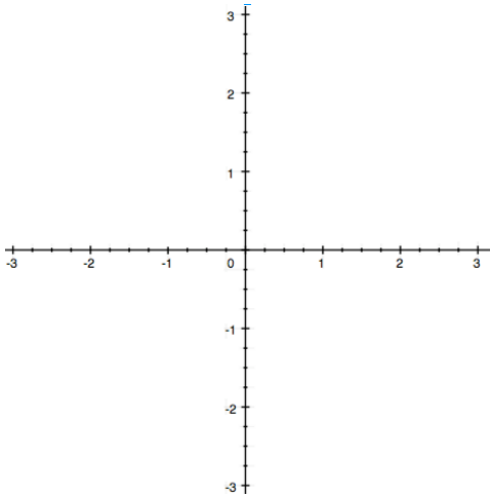


FIGURA 1.2.5.1 Sistema coordenado rectangular

Las coordenadas se consideran que son los números reales en la matemática elemental, pero pueden ser números complejos o elementos de un sistema más abstracto. El uso de un sistema de coordenadas permite que los problemas en la geometría se traduzcan en problemas acerca de los números y viceversa; esta es la base de la geometría analítica.

1.2.5.1 Sistema de Coordenadas Cartesianas

El ejemplo prototípico de un sistema de coordenadas es el sistema de coordenadas cartesianas. Estas son dos líneas perpendiculares donde se toman las coordenadas de un punto según las distancias con las líneas teniendo en cuenta los signos de ésta.

En tres dimensiones, son tres planos perpendiculares donde se eligen las tres coordenadas de un punto y son las distancias con signo a cada uno de los planos. Esto se puede generalizar para crear n coordenadas de cualquier punto en n-dimensiones en el espacio euclidiano.

Dependiendo de la dirección y el orden del eje de coordenadas del sistema puede ser un derecho o un sistema de la izquierda. Este es uno de los muchos sistemas de coordenadas.

1.2.5.2 Sistema de Coordenadas Cilíndrico

Un sistema cilíndrico de coordenadas es un sistema de coordenadas tridimensional que especifica posiciones de puntos por la distancia desde un eje de referencia elegido, la dirección desde el eje con respecto a una dirección de referencia elegido y la distancia desde un plano de referencia elegido perpendicular al eje. Esta última distancia se da como un número positivo o negativo dependiendo de qué lado del plano de referencia se enfrenta el punto.

El origen del sistema es el punto donde las tres coordenadas se pueden dar como cero. Esta es la intersección entre el plano de referencia y el eje. El eje se llama indistintamente cilíndrica o eje longitudinal, para diferenciarlo del eje polar, que es el rayo que se encuentra en el plano de referencia, comenzando en el origen y que apunta en la dirección de referencia.

1.2.5.3 Sistema de Coordenadas Esféricas

Un sistema de coordenadas esféricas es un sistema de coordenadas para un espacio tridimensional, donde la posición de un punto es especificado por tres números: la distancia radial de dicho punto a partir de un origen fijo, su ángulo polar medido desde una dirección cenit fijo, y el ángulo de azimut de su proyección ortogonal sobre un plano

de referencia que pasa a través del origen y es ortogonal al cenit, medida a partir de una dirección de referencia fijo en el plano.
La distancia radial también se llama el radio o radial de coordenadas. El ángulo polar puede ser llamado colatitud, ángulo cenital, ángulo normal, o ángulo de inclinación.

1.2.6 Punto

Un punto se refiere generalmente a un elemento de un conjunto llamado un espacio.

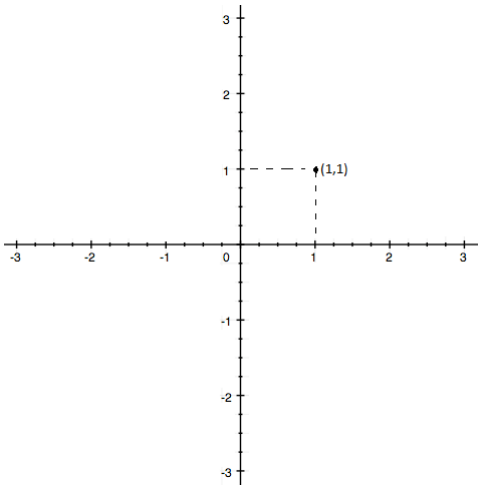


FIGURA 1.2.6.1 Sistema coordenado rectangular

Más específicamente, en la geometría euclidiana, un punto es una noción primitiva sobre la que se construye la geometría. Al ser una noción primitiva significa que un punto que no se puede definir en términos de objetos definidos previamente, es decir, un punto se define solamente por algunas propiedades, llamadas axiomas, que debe satisfacer. En particular, los puntos geométricos no tienen longitud, superficie, volumen, o cualquier otro atributo dimensional. Una interpretación común es que el concepto de un punto está destinado a captar la noción de una ubicación única en el espacio euclidiano.

Puntos, considerados en el marco de la geometría euclidiana, son uno de los objetos más fundamentales. Euclides definió originalmente el punto como "lo que no tiene ninguna parte". En el espacio euclidiano de dos dimensiones, un punto está representado por un par ordenado (x, y) de los números, en el que el primer número representa

convencionalmente la horizontal y con frecuencia se denota por x, y el segundo número representa convencionalmente la vertical y a menudo se denota por y. Esta idea se generaliza fácilmente al espacio euclidiano tridimensional, donde un punto es representado por un triplete ordenado (x, y, z) con el tercer número adicional que representa la profundidad y a menudo denota por "z". Otras generalizaciones están representados por un grupo irregular ordenada de n términos, (a1, a2, ..., an), donde n es la dimensión del espacio en el que se encuentra el punto.

1.2.7 Vectores

Un espacio vectorial (también llamado un espacio lineal) es una colección de objetos llamados vectores, que pueden añadirse juntos y multiplicarse por números, llamados escalares. En este contexto, los escalares se toman a menudo para ser números reales, pero también hay espacios vectoriales con la multiplicación escalar por los números complejos, los números racionales, o en general cualquier campo. Las operaciones de adición y multiplicación escalar de vectores deben satisfacer ciertos requisitos, llamados axiomas.

Los vectores euclidianos son un ejemplo de un espacio vectorial. Ellos representan cantidades físicas tales como las fuerzas: dos fuerzas (del mismo tipo) pueden ser añadidos para proporcionar un tercero, y la multiplicación de un vector de fuerza por un multiplicador real es otro vector de fuerza. En la misma línea, pero en un sentido más geométrico, vectores que representan desplazamientos en el plano o en el espacio tridimensional también forman espacios vectoriales. Vectores en espacios vectoriales no necesariamente tienen que ser objetos como una flecha, tal como aparecen en la Figura 1.2.7.1 en la parte de abajo: los vectores son considerados como objetos matemáticos abstractos con propiedades particulares, que en algunos casos pueden ser visualizados como flechas.

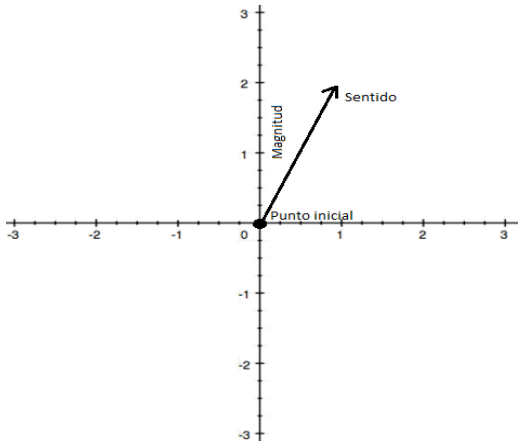


FIGURA 1.2.7.1 Vector euclidiano

Tomemos por ejemplo, un espacio vectorial sobre un campo F, un conjunto V junto con dos operaciones que cumplen los ocho axiomas enumerados. Los elementos de V son comúnmente llamados vectores. Elementos de F son comúnmente llamados escalares. La primera operación, denominada suma de vectores o simplemente suma, toma dos vectores “v” y “w”, y les asigna un tercer vector que comúnmente se escribe como v + w, y llama la suma de estos dos vectores. La segunda operación, denominada multiplicación escalar toma cualquier escalar “a” y cualquier vector “v” y da otro vector av.

1.2.8 Normal

Una normal es un objeto tal como una línea o vector que es perpendicular a un objeto dado, como se muestra en la figura 1.2.8.1. Por ejemplo, en el caso de dos dimensiones, la línea normal a una curva en un punto dado es la línea perpendicular a la línea tangente a la curva en el punto.

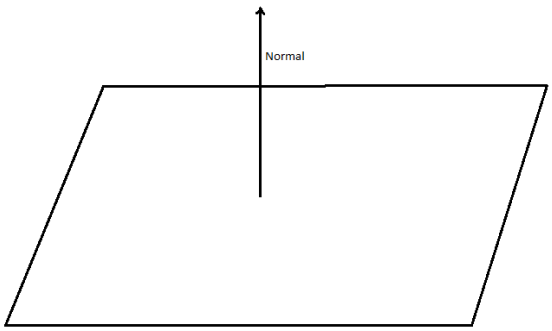


FIGURA 1.2.8.1 Normal de un plano

En el caso de tres dimensiones de una superficie normal, o simplemente normal a una superficie en un punto P es un vector que es perpendicular al plano tangente a la

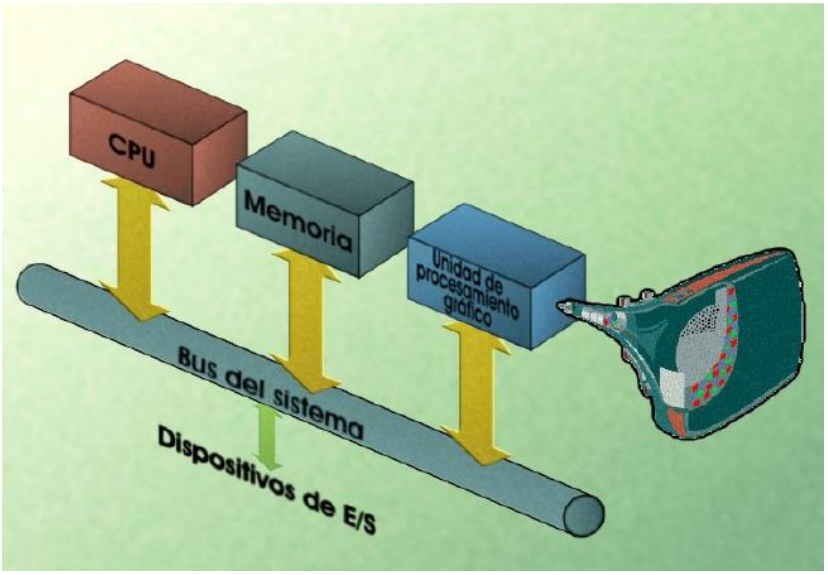
superficie en P. La palabra "normal" también se usa como un adjetivo: una línea normal a un plano, la componente normal de una fuerza, el vector normal, etc. el concepto de normalidad se generaliza a la ortogonalidad.

El concepto se ha generalizado a las variedades diferenciables de dimensión arbitraria incrustados en un espacio euclidiano. El espacio de vector normal o el espacio normal de un colector en un punto P es el conjunto de los vectores que son ortogonales al espacio tangente en P. En el caso de curvas diferenciales, el vector de curvatura es un vector normal de especial interés.

La normal se utiliza a menudo en gráficos por ordenador para determinar la orientación de una superficie hacia una fuente de luz para el sombreado plano, o la orientación de cada una de las esquinas (vértices) para imitar una superficie curva con sombreado de Phong.

1.3 Principios de graficación por computadora

Un sistema grafico tradicional consta de seis componentes: procesador, memoria, unidad de procesamiento gráfico, bus del sistema dispositivos de entrada y dispositivos de salida. El procesador desempeña un papel central en cualquier sistema gráfico y los demás componentes deben comunicarse en algún momento con otro, incluso con el procesador, mediante el bus de datos o canal de datos, generalmente los dispositivos de salida de un sistema grafico es un monitor de video, los más comunes son los que emplean tecnología LCD, aunque también existen los de tecnología CRT que son de menor calidad de video y los de tecnología LED que son los de mejor definición en la actualidad, pero debido a su costo se siguen utilizando los LCD.



1.3.1 Diagrama de un sistema grafico[3]

Procesador. Es el hardware dentro de una computadora u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema, es el cerebro de la computadora, dependiendo del tipo de procesador y su velocidad determinará un mejor o peor rendimiento, hoy en día existen varias marcas y tipos, sin embargo, para la generación de gráficos asistidos por computadora se necesita un procesador de gama alta, para que sea capaz de procesar imágenes en el menor tiempo posible[4].

Memoria. La memoria es el dispositivo que retiene o almacena información durante algún intervalo de tiempo, la memoria proporciona una de las principales funciones de la computación moderna: el almacenamiento de información y conocimiento. Es uno de los componentes fundamentales de la computadora, que interconectados con el procesador y los dispositivos de entrada/salida, implementan las funciones fundamentales de la arquitectura de von Neumann.

[3] http://acercadecomputaciongrafica.blogspot.mx/2011/04/el-hardware-grafico.html Blogspot
[4] Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

Unidad de Procesamiento Grafico (GPU). Es un coprocesador dedicado al procesamiento de gráficos u operaciones para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos, con la GPU se pueden emplear primitivas para dibujar rectángulos, triángulos, círculos y arcos actualmente disponen de gran cantidad de primitivas, buscando mayor realismo en los efectos.

Bus del Sistema. El ***bus*** (canal) es un sistema digital que transfiere datos entre los componentes de una computadora o incluso entre varias computadoras, está formado por cables o pistas en un circuito impreso, además, de circuitos integrados.

Dispositivo de entrada. Es cualquier periférico utilizado para proporcionar datos y señales de control a un sistema de procesamiento de información.

Dispositivo de salida. Es aquel periférico que recibe información de la computadora.

Hardware gráfico. Como se mencionó anteriormente, los dispositivos de salida empleados en la generación de gráficos asistidos por computadora son muy importantes y dentro de los más usados son los monitores de tecnología LCD y los de tecnología LED

Tecnología LCD. Los monitores LCD están compuestos por cristales líquidos, son sustancias transparentes con cualidades propias de líquidos y de sólidos, una pantalla LCD está formada por dos filtros polarizantes, uno horizontal y uno vertical con filas de cristales líquidos alineados perpendicularmente entre sí, de modo que al aplicar o dejar de aplicar una corriente eléctrica a los filtros, se consigue que la luz pase o no pase a través de ellos, dependiendo de cuanto bloquee el segundo filtro el paso de la luz que ha atravesado el primero. El color se consigue añadiendo tres filtros adicionales de color uno rojo, uno verde y uno azul sin embargo, para la reproducción de varias tonalidades de color, se deben aplicar diferentes niveles de brillo intermedios entre luz y no-luz, lo cual se consigue con variaciones en el voltaje que se aplica a los filtros.

Un monitor LCD cuenta con cinco componentes principales: reflectores de luz o lámparas, acetatos o difusor de luz, pantalla de cristal líquido, tarjeta de video y la fuente de poder.

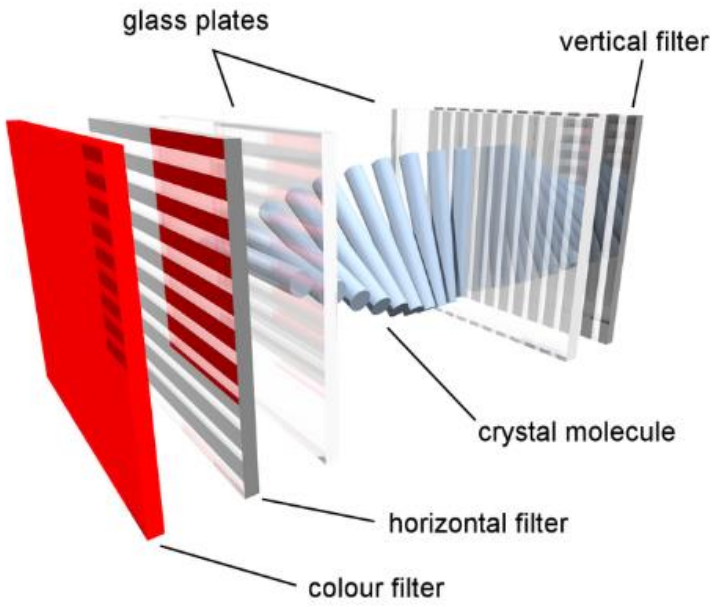
Reflectores de luz o lámparas. Son los componentes que generan una luz blanca que con los filtros se vuelve una luz colorida con cada pixel la pantalla consta de cuatro lámparas fluorescentes que están ensambladas en pares ubicados en la parte superior e inferior de la pantalla

Acetatos o difusor de luz. Permite que la luz blanca salga como una luz coloreada, distribuyendo la iluminación uniformemente por toda la pantalla.

Pantalla de cristal líquido. Por sus siglas en inglés LCD (liquid cristal display), proporciona la calidad y la definición del monitor que se esté iluminando.

Tarjeta de video. Se encuentra integrada a la pantalla y es la encargada de recibir y traducir los datos de la CPU, para procesarlos y trasmitir la imagen en la pantalla.

Fuente de poder. Es la encargada de convertir la corriente AC en corriente DC para permitir el funcionamiento del monitor.



1.3.2 Diagrama de un monitor LCD[5]

[5] https://es.wikipedia.org/wiki/Pantalla_de_cristal_l%C3%ADquido#/media/File:LCD_subpixel_(en).png Wikimedia

Tecnología LED. Una pantalla de LED es un dispositivo electrónico conformado por un arreglo de diodos emisores de luz o LEDs, que puede desplegar datos, información, imágenes y vídeos a los usuarios.

Dentro de la tecnología de monitores de emisión de luz, los LED's se han situado entre los dispositivos más utilizados, su nombre proviene de las siglas del inglés Light-Emitting Diode, o diodo de emisión de luz, los LED's son considerados diodos debido a que permiten las circulación de la energía eléctrica en una sola dirección y tienen dos terminales.

Los LED's trabajan a través de la electroluminiscencia, es decir, aprovechan el fenómeno óptico y eléctrico donde un material emite luz como respuesta al flujo de una corriente eléctrica a través de él, cuando la energía eléctrica fluye a través de los LED's se produce luz.

Por lo general, los LED's son rojos, azules y verdes y el color de la luz que emiten depende de la banda de energía del semiconductor. Ver figura 1.3.3.

Las partes que componen un LED son:

– Ánodo y cátodo

– Lente o encapsulado epóxido

– Contacto metálico

– Yunque

– Semiconductor

– Plaqueta

– Reflector

– Borde plano

Figura 1.3.3 Monitor LED[6]

Un pixel es la unidad mínima de representación gráfica y que se puede mostrar en un monitor, la resolución de una imagen es el indicador de la calidad gráfica y se determina por el número de pixeles en una determinada área, al cálculo que se realiza para definir de qué color se iluminara cada pixel de nuestro monitor se le conoce como proceso de barrido[7], es tan rápido que el ojo humano no alcanza a distinguir como se activan los puntos por separado percibiendo la ilusión de que todos los pixeles se iluminan al mismo tiempo.

Por otra parte la memoria de video almacena el resultado de las funciones gráficas, cada pixel en pantalla corresponde a una entrada en particular en un arreglo bidimensional en memoria, el número de renglones en el arreglo de memoria es igual al número de pixeles en cada línea de barrido.

El termino pixel también se usa para describir el renglón y la columna de colocación en el arreglo de la memoria de video que corresponde a la posición en pantalla. Cada pixel se codifica mediante un conjunto de bits de longitud determinada conocida como llamada de profundidad de color, por ejemplo, puede codificarse un pixel con un byte, de manera

[6] http://es.slideshare.net/jmtoledodiaz/tipos-de-monitores-josemanueltoledodiaz1smr Slide Share

[7] Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

que cada pixel admite 256 caracteres variantes, dos dígitos por bit elevados a la octava potencia. En las imágenes de color verdadero se suelen usar tres bytes para definir un color, es decir, en total podemos representar un total de 2 elevado a la 24 potencia, o sea, 16, 777,216 colores diferentes[8].

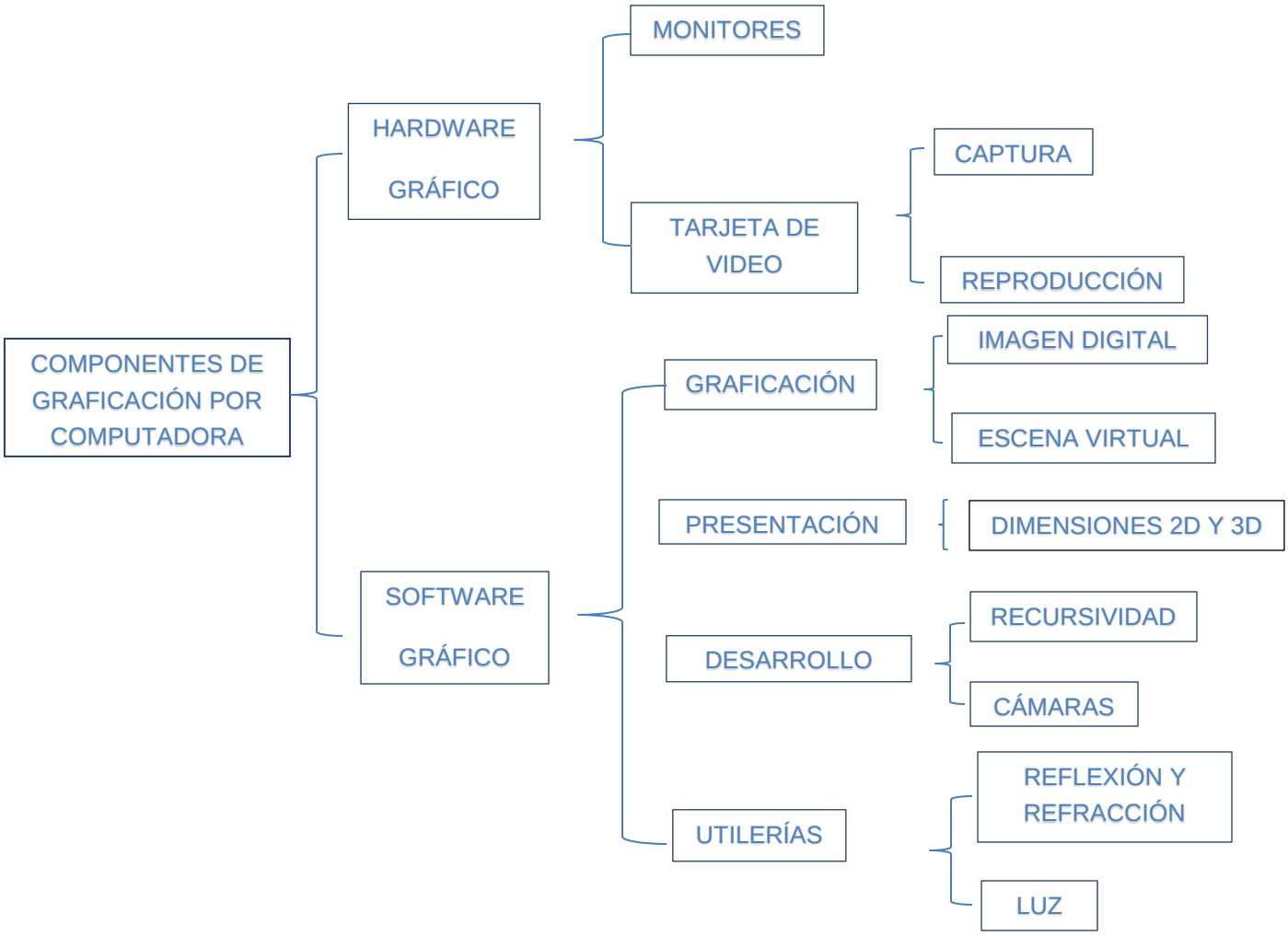
El controlador de video es un dispositivo de hardware que lee el contenido de la memoria de video y lo deposita en un buffer de video, para luego convertir la representación digital de una cadena de valores de pixeles en señales analógicas de tensión que se envían en serie a la pantalla de video, el controlador de video lee de la memoria de video el color del pixel a dibujar y envía la información a un convertidor digital analógico en donde por medio de las componentes de color RGB (verde, rojo y azul) se determina la intensidad del pixel.

Software gráfico: Por lo general los desarrolladores utilizan una serie de librerías de programación gráfica que le permiten escribir aplicaciones sin tener que llegar a conocer en detalle el hardware sobre el que se ejecutará su código, ejemplos de estas librerías podrían ser OpenGL de SGI y Direct3D de Microsoft.

Por lo general, estas librerías permiten trabajar creando estructuras en un sistema de coordenadas local, integrar estas estructuras en una escena más compleja que utiliza un sistema de coordenadas global, el software transformará estas coordenadas a unas coordenadas de dispositivo normalizado y en un último paso estas se ajustarán al rango de salida del dispositivo final.

En el cuadro 1.3.4 se muestra en resumen los principios para generar gráficos por computadora, así como se aprecian componentes de software y hardware y los resultados o elementos que se obtiene de cada uno.

[8] Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014



Cuadro 1.3.4 Principios de graficación por computadora

1.4 Fundamentos del Raytracer

El R*aytracing* (trazado de rayos), es una técnica utilizada frecuentemente en la computación gráfica para la generación de imágenes basadas en escenas tridimensionales. Aunque este es su uso más común, el algoritmo se utiliza en muchos campos dentro de la ingeniería en computación, principalmente en simulaciones de fenómenos físicos, como es la luz en el caso particular de la graficación por computadora.

La primera vez que se utilizó para la representación de imágenes fue en 1968 por Arthur Appel[9], a la presentación original de la técnica se le nombró *raycasting* (emisión de rayos) pues sólo planteaba por cada rayo obtener información del objeto más próximo a la cámara, obteniendo la información de sus propiedades y material exclusivamente, con este método es imposible representar superficies reflejantes o transparentes, así como sombras. Fue hasta 1979 que Turner Whitted[10] presentó el algoritmo como se le conoce hoy en día, en lugar de sólo obtener información del objeto más próximo en la emisión del rayo, este revisa las propiedades del objeto en cuestión y de ser necesario crea nuevos rayos a partir del punto de colisión con el objeto dependiendo de su material pudiendo así reproducir reflexiones, refracciones o sombras. La naturaleza de esta técnica convirtió el ciclo principal del algoritmo en una función recursiva capaz de seguir generando rayos conforme fuesen necesarios hasta cumplir con las condiciones de la imagen y fue llamado raytracing.

La principal aplicación del raytracing y su uso en la generación de imágenes por computadora, es tratar de replicar el mundo real simulando cómo se comporta la luz en las superficies de distintos objetos, y tratar de aproximarse lo más posible a una fotografía. Sin embargo, el raytracing funciona distinto que la visión humana, el ojo percibe la luz reflejada en objetos de distintos materiales, los cuales absorben longitudes de onda del espectro visible, y finalmente vemos una imagen con colores, reflejos y sombras. Tratar de replicar el comportamiento de la luz de esta manera sería imposible

[9] Some techniques for machine rendering of solids - http://graphics.stanford.edu/courses/Appel.pdf
[10] An Improved Illumination Model for Shaded Display - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.1534

con una computadora, partir desde la fuente de luz hasta encontrar nuestro camino hacia la cámara requeriría de una infinita cantidad de muestreo que probablemente nunca llegaran a la imagen final. Es por eso que en el raytracing usamos la cámara como punto de partida de los rayos de luz y trazamos su camino hasta el origen de la fuente de luz, tomando muestras de color en el camino hasta obtener el color final.

Para ejemplificar cómo funciona la técnica imaginemos una cámara estenopéica, esto es una caja con un pequeño orificio al frente y un material fotosensible (filme) en la parte posterior interior de la caja. La luz entra por el orificio y es absorbida por el material produciendo una imagen invertida de lo que se desea fotografiar.

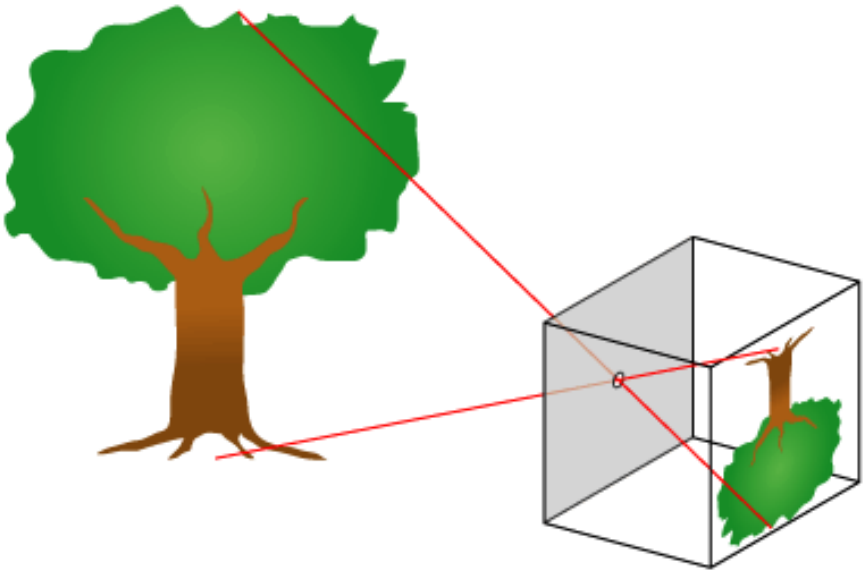


Figura 1.4.1 Cámara Estenopeica[11]

Podemos observar en la figura 1.4.1 que la distancia del filme al orificio está relacionada directamente con el campo de visión de la imagen que queremos fotografiar, este es el foco de nuestra cámara, y en cámaras modernas esta es la distancia a la que se encuentra el lente del sensor o filme.

[11] http://commons.wikimedia.org/wiki/Image:Pinhole-camera.png – Wikimedia Commons – Public Domain

Para simular el flujo de la luz hacia el filme comenzamos con una cámara virtual, la cual se encuentra definida dentro de la descripción de la escena que queremos reproducir. Esta cámara tiene una posición y una dirección hacia la que está viendo, tomaremos el primero como el origen desde donde trazar todos nuestros rayos, y posteriormente se crea una matriz perpendicular al vector formado por la cámara (posición → dirección), este será nuestro filme, la distancia entre el origen y la matriz será nuestro foco o campo de visión y el tamaño de la matriz lo define la resolución de la imagen que queremos obtener, cada posición de la matriz será un pixel a través del cual trazaremos uno o más rayos hacia la escena. Cabe mencionar que el tiempo de trazado escala proporcionalmente con el tamaño de la imagen, cada pixel extra que queramos de resolución agregara tiempo para calcular el resultado final.

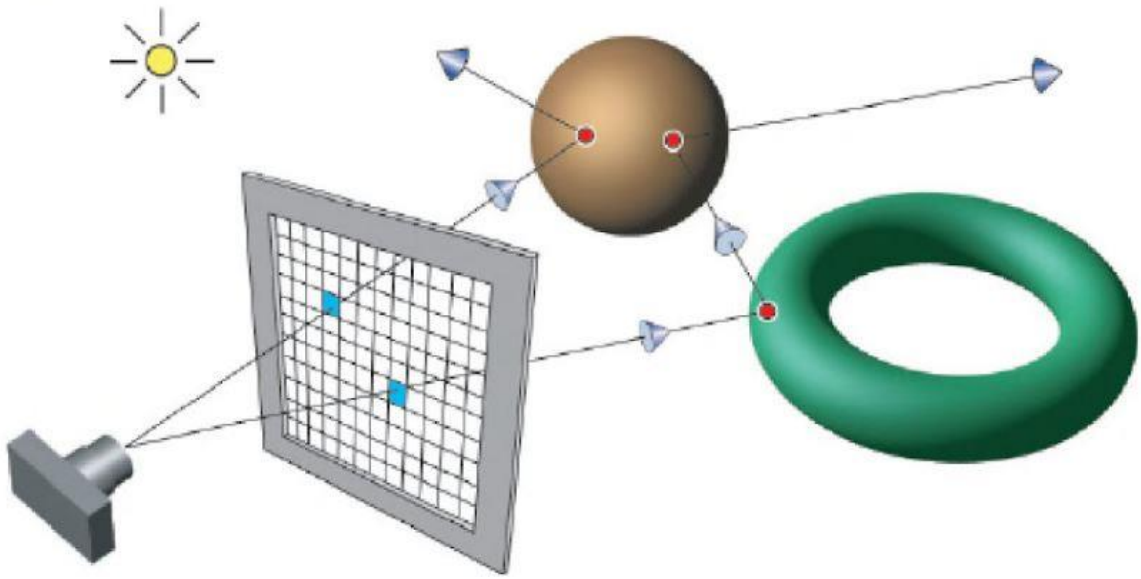


Figura 1.4.2 Trazado de rayos[12]

Se puede observar en la figura 1.4.2 la cámara así como la matriz o imagen a través de la cual trazaremos los rayos hacia la escena, los objetos que se encuentran en ella y las fuentes de luz existentes. Se pueden llegar a trazar más de un rayo por pixel con el fin de obtener más muestras de color y promediar así un mejor resultado.

[12] Ray Tracing from the Ground Up – Kevin Suffern, 2007.

El ciclo principal del algoritmo traza rayos desde el origen de la cámara a través de cada uno de los pixeles de nuestra imagen, para obtener el resultado, cada rayo revisa colisiones con cada uno de los objetos de la escena. Distintos tipos de objeto tienen una función de colisión diferente, con la cual se revisa si el rayo colisiona con el objeto, así como la distancia a la que fue la colisión. Después de revisar colisiones con todos los objetos, se selecciona aquel con la distancia más cercana de colisión, esto significa que es el objeto que ve la cámara directamente, se revisan sus propiedades como color y material y se obtiene el color final del pixel que tendrá la imagen en ese punto. Dependiendo de las características del objeto, se calcula el color basado en un *shader* (técnica de sombreado) con el cual se logran diferentes tipos de iluminación sobre la superficie del objeto dependiendo de su material.

Existen shaders para simular materiales metálicos o plásticos, ya que la luz se dispersa distinto sobre la superficie de cada uno, principalmente hay dos componentes para el sombreado de la superficie de un objeto, el difuso y el especular, y tienen que ver con el comportamiento de la luz en dicha superficie. Como se mencionó previamente, el raytracing es un método recursivo y es aquí donde se utiliza la recursividad del algoritmo para seguir disparando rayos dependiendo de las características del objeto.

En el caso de materiales como espejos, metales, o vidrio, se necesita de más información para obtener el color final del pixel para el que se están trazando los rayos. Es el caso de reflexión, se traza recursivamente un rayo perpendicular al punto de colisión con la superficie, buscando nuevamente entre todos los objetos de la escena hasta obtener el más cercano al punto, pues es dicho objeto el que se verá reflejado en el material. En caso de encontrar otro objeto con reflexión, se seguirán trazando rayos de la misma manera, es importante establecer un parámetro fijo de profundidad de trazado, ya que puede darse el caso donde se sigan trazando rayos infinitamente, como el de dos espejos encontrados, y es necesario para detener el proceso. Para materiales traslúcidos como vidrio o líquidos, que también refractan la luz en el medio, se trazan rayos desde el punto de colisión en la superficie a través del objeto utilizando el *IOR* (índice de refracción) del material, se buscan colisiones nuevamente con todos los objetos de la escena con el fin de obtener el color final en este punto.

Para el cálculo de sombras entre objetos, se tienen que definir lámparas o fuentes de luz que las provoquen. Hay distintos tipos, sin embargo, las dos más comunes son la luz ambiental y la luz puntual. La luz ambiental en nuestro caso no provoca sombras, es simplemente un valor de iluminación que se suma a cada color de cada pixel trazado, esto con el fin de manipular la iluminación general de la escena, este tipo de luz no provoca sombras pues no parte de un punto en específico. Por otro lado, la luz puntual necesita de un vector de posición así como un valor de intensidad de iluminación de la lámpara. Una vez encontrada la colisión con un objeto, se traza un rayo desde el punto de colisión con la superficie, hacia cada fuente de luz puntual de la escena, si este rayo no colisiona con ningún otro objeto, quiere decir que la fuente de luz, ilumina directamente este punto, en caso de haber colisiones, significa que hay otros objetos bloqueando directamente la fuente de luz y procedemos a generar una sombra. Esto último se hace para cada fuente de luz en nuestra escena y para cada rayo que encuentra una colisión con un objeto, pudiendo así generar múltiples sombras a partir de distintas lámparas.

El procedimiento recursivo del algoritmo y el trazado de rayos, donde por un solo pixel realmente se emiten múltiples rayos dependiendo de las características del material del objeto.

Aunque el Raytracing es una técnica simple, se ha extendido en una infinidad de formas para lograr resultados más realistas, desde la creación de nuevos shaders y técnicas avanzadas de iluminación global que simulan una iluminación similar a la de la atmósfera sobre la escena. Se pueden aplicar texturas a los objetos extrapolando las coordenadas de una imagen sobre su superficie así como aplicar mapas de alturas para obtener materiales con relieve o superficies rugosas, sin la necesidad del costo computacional que requeriría modelarlos. La utilización de *oversampling* (sobre muestreo) para obtener líneas más finas y mejorar la calidad de la imagen final, es algo muy común, pero a su vez, agrega costo computacional al algoritmo teniendo que trazar magnitudes superiores de rayos por pixel. Pero sin lugar a dudas, los mayores avances que se le han hecho al algoritmo tienen que ver con su optimización. Cómo es de esperarse, el raytracing es un proceso costoso, requiere de mucho procesamiento trazar miles de millones de rayos en una escena compleja, y esto se hace miles de veces en una sola imagen.

Es por esto que se han creado técnicas para agilizar el proceso, una muy común es la de utilizar estructuras de datos y árboles para subdividir la escena espacialmente y así hacer menos chequeos de colisiones con objetos, pero debido a la naturaleza del algoritmo, la implementación de procesamiento paralelo es probablemente la más utilizada. El costo computacional del algoritmo se hace un pixel a la vez, pudiendo así utilizar más recursos de hardware para calcular múltiples pixeles de manera simultánea y obtener el resultado más rápidamente. Hoy en día, la industria cinematográfica utiliza granjas con miles de procesadores con el fin de agilizar el proceso de renderizado de imágenes por computadora.

1.5 Diagrama funcional del algoritmo

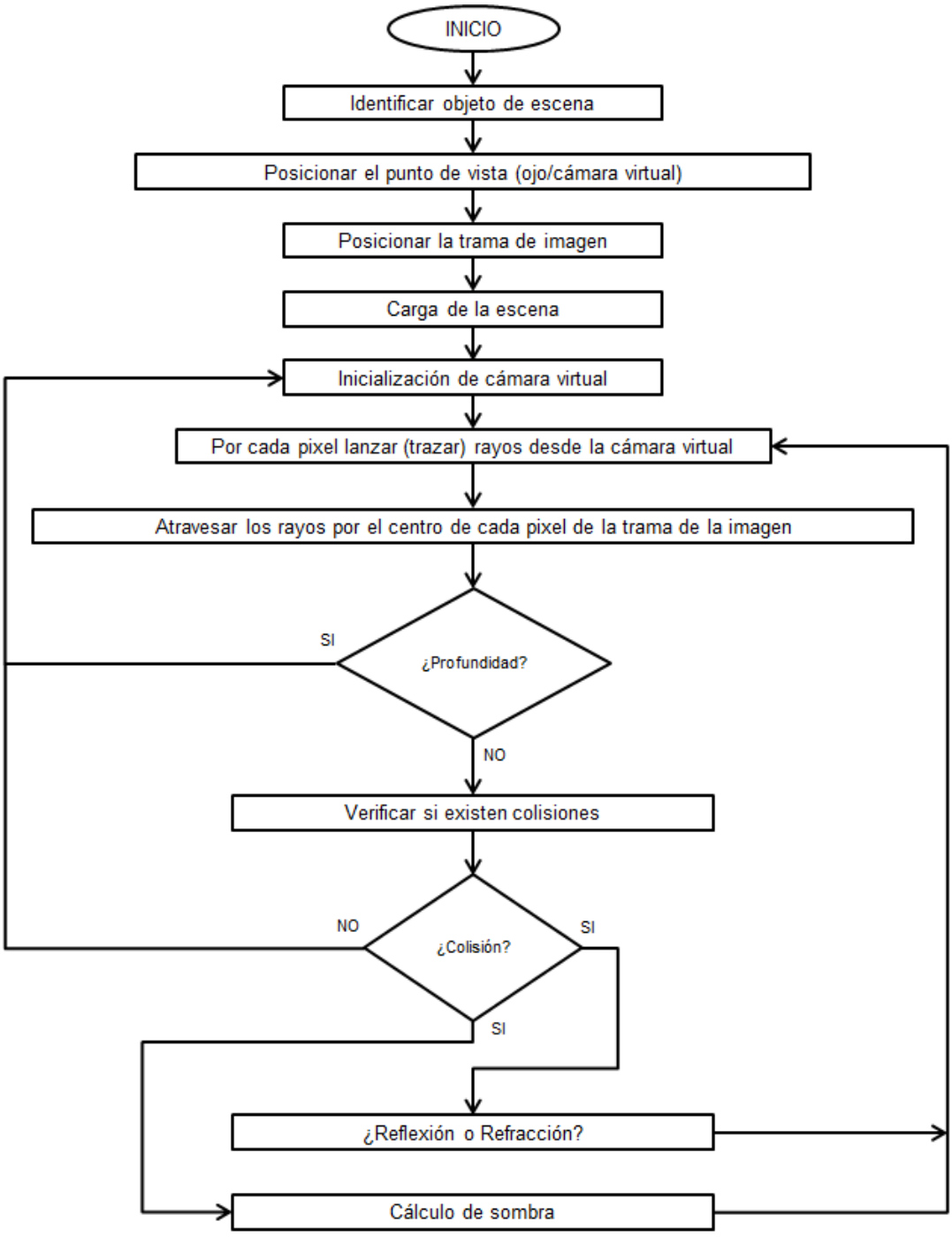


FIGURA 1.5 Diagrama funcional [13]

[13] http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview

1.6 Alcance y acotamiento del problema

El aprendizaje y la enseñanza son dos procesos muy importantes dentro de cualquier institución educativa debido a que estos hacen que el individuo adquiera conocimientos y desarrolle habilidades. El tener individuos con capacidades y habilidades significativas en cuanto a conocimiento hace que progrese una sociedad. Existen diferentes tipos de enseñanzas y todos tienen sus ventajas y desventajas como cualquier cosa en esta vida. Pero sin lugar a dudas algo que tienen en común los métodos de enseñanza que funcionan, es que van de lo simple a lo complejo de una forma estructurada y orgánica. Esto permite al alumno ir asimilando la información poco a poco pero de una manera continua, integrándola de la mejor manera dentro de sus estructuras cognitivas de aprendizaje.

En el esquema demasiado tradicional de enseñanza, toda la teoría primero y sólo al final realizar algún proyecto, donde la aplicación del conocimiento se exprese, tiene la desventaja de que no se cuantifica cuánto del conocimiento fue asimilado por el alumno, no se dé una forma detallada en que temas el alumno falló, y que temas si aprendió. Simplemente se da por descontado una de las dos opciones, aprendió o no aprendió. Esta visión binaria para temas de aprendizaje puede ser bastante peligrosa debido a que no establece matices. El saber de una manera detallada dónde el alumno falla permite implementar mejores prácticas educativas. Otro problema es que la distribución del tiempo de enseñanza no está ponderada de una buena manera, ya que puede suceder que a temas sin demasiada relevancia se le dedique mucho tiempo, o que por el contrario a temas centrales se les dedique poco tiempo. Además existen diferentes tipos de alumnos con diferentes niveles de asimilación de los temas. Algunos asimilarán más rápido que sus contrapartes. Mucho del aprendizaje de un estudiante de ingeniería no se da en el aula, sino en las horas que pasa fuera, ya sea en las bibliotecas, laboratorios, en casa realizando trabajos y proyectos. Debido a esto es muy importante tener herramientas que permitan completar el trabajo que se realiza en el aula. Que estas herramientas sean fáciles de entender y que le permitan asimilar la mayor cantidad de conocimiento posible. Además que tengan formas de autodiagnóstico, donde el alumno se puede dar cuenta en que temas falla y como solucionarlos.

La herramienta de las prácticas del Raytracer soluciona muchos de los problemas anteriormente planteados. Además permite la fácil implementación en un laboratorio dentro de la institución, debido a que están estructuradas de tal forma que el alumno vaya de lo básico a lo avanzado, comprendiendo y aplicando los conceptos de la computación gráfica.

La teoría sin práctica está condenada a ser una forma bastante deficiente de aprendizaje, debido a que el conocimiento no se integra totalmente. Y aunque la teoría sola sin práctica pueda ser una forma valida de aprender para algunas áreas del conocimiento. Esto no aplica para la Ingeniería, en este caso, de la Ingeniería en Computación. Cuando la labor de ésta es aplicar el conocimiento de las ciencias teóricas (Física y matemáticas). Es parte intrínseca de la Ingeniería la aplicación del conocimiento. La forma de aplicarlo gradual y controladamente es la realización de prácticas. De ahí el uso del formato (Teoría y Prácticas), debido a que este formato permite que el conocimiento se aprenda de forma teórica y después se lleve a la implementación, creando un programa en donde se ponen en marcha los conceptos que se han aprendido en el apartado de teoría.

Las prácticas están diseñadas para que el alumno de octavo semestre de la materia Computación Gráfica e Interacción humano-computadora del plan de estudios 2016 de la carrera de Ingeniería en Computación en la Facultad de Ingeniería, tenga todos los elementos para programar un Raytracer. Con lo cual demostraría un conocimiento aplicado de la computación gráfica, debido a que la construcción de un Raytracer involucra desde aspectos básicos hasta temas más avanzados, integrándolos de una forma natural en la implementación del algoritmo.

Los algoritmos son de gran utilidad en el campo de la ingeniería debido a que son un método (secuencia de pasos estructurados) que da solución a un problema de una forma óptima. Se pueden usar para resolver problemas sin que se tenga que reinventar la rueda. Raytracing al ser el algoritmo que permite crear imágenes en 3D en un medio de dos dimensiones (la pantalla del monitor), nos brinda una solución a ese importante problema.

El desafío se encuentra en cómo implementarlo, creando en el alumno el conocimiento y las habilidades necesarias.

De ahí que se haya partido por usar la vieja "técnica" *divide y vencerás*, dividiendo el algoritmo en sus unidades básicas, y a partir de estás enseñar los conceptos que se encuentran detrás para más adelante generar el seudocódigo necesario, a partir de éste que el alumno pueda codificar el algoritmo Raytracer, de una manera que le permita integrar lo que ha aprendido en la parte teórica.

Debido a que las prácticas están estructuradas en un modelo progresivo, el alumno al realizarla, le servirá como base de la siguiente y así sucesivamente. Hasta que al final de todas ellas tenga un producto bastante robusto que involucra todos los temas y conceptos necesarios para la construcción del Raytracer.

La herramienta didáctica y educativa de las prácticas del Raytracer no pretende sustituir ningún otro material, libro, tutorial, programa o cualquier otro tipo de enseñanza. Simplemente es una contribución más para que el aprendizaje de la ingeniería cuente con más recursos que le resulten atractivos a los estudiantes de Ingeniería en Computación. Al ser de una forma que cuenta con muchos aspectos que hacen que las prácticas del Raytracer puedan ser llevadas incluso desde un enfoque autodidacta, que en la actualidad es sumamente útil y necesario.

Para aprobar cada una de las prácticas Raytracer, el alumno y/o profesor, tiene que cerciorarse de que ésta sea ejecutada de manera correcta a partir de un conjunto de pruebas para demostrar que la implementación se realizó adecuadamente. El código es el correcto, realiza las funciones que se detallan en el seudocódigo. Esto permite que todas las partes de la creación del Raytracer se realicen adecuadamente. En caso de que el código no pase las pruebas, el problema se soluciona de una manera relativamente sencilla. El alumno repasa el tema de teoría que también se incluye como parte de la práctica Raytracer. Después revisa el seudocódigo, que lo haya implementado adecuadamente. Ésta es una manera muy buena de trabajar debido a que los fallos están

acotados a un universo pequeño, y no como en algunos otros casos de desarrollos en donde el programador no sabe dónde se encuentran los posibles errores, lo cual hace complicado el desarrollo del proyecto. Utilizando la metodología Incremental Build Model (Modelo de Construcción Incremental IBM) que permite hacer que el desarrollo de una aplicación sea lo más ordenado y estructurado posible. La metodología IBM es de fácil implementación para el alumno, él ya que ira armando pequeños módulos del programa practica con práctica, es decir siguiendo las instrucciones de las prácticas el alumno con su código desarrollara modulo a modulo un programa que pueda generar un raytracer bastante robusto y funcional ya que al ser seccionado en diferentes temas realizara diferentes pruebas y al final ya no presentara ningún problema el programa desarrollado durante el curso. Lo cual como ventaja adicional le va dando una estructura mental de programación que le será muy útil en proyectos posteriores. Fortaleciendo y creando otra habilidad muy útil para el estudiante y el profesional de la Ingeniería en Computación.
El alumno que realice las prácticas deberá tener conocimientos previos de geometría analítica, dominar el álgebra vectorial; la ecuación del plano y de la esfera, la intersección entre superficies. Conocimientos de trigonometría y física general. Aunque en las prácticas se incluye la parte teórica sobre estos temas, es importante que el alumno ya haya estudiado esos temas, para asimilar más rápido el conocimiento de la computación gráfica. De todas formas la parte teórica de las prácticas Raytracer está estructurada para que incluso con deficiencias en las ciencias básicas de la Ingeniería, se pueda aprender con la sección teórica de las prácticas. El alumno será capaz de entender y asimilar los conocimientos.

Es indispensable para poder realizar las prácticas Raytracer, dominar un lenguaje de programación ya sea de programación estructurada u orientada a objetos. Capacidad para interpretar seudocódigo. La herramienta no es para aprender a programar en un lenguaje, ni es para realizar seudocódigo. Es para codificar (programar) el seudocódigo que se encuentra en las prácticas, el seudocódigo proporcionado ya está armado con los conocimientos básicos de la computación gráfica.

Las prácticas Raytracer brindarán un conocimiento suficiente del algebra vectorial aplicado a la computación gráfica debido a que esas operaciones son fundamentales en el algoritmo del raytracer. Se definen en las prácticas conceptos importantes como el de objetos y primitivas, así como el color que es expresado en un monitor de computadora; como se construye una imagen digital; que es una escena virtual y como está compuesta; como es el proceso de trazado de rayos, que es ahí donde radica el poder y simplicidad del algoritmo Raytracing.

En las imágenes 3D es muy importante los temas de las luces y las sombras ya que hacen que la imagen tenga perspectiva, lo cual da la apariencia de tercera dimensión aún y cuando sea una imagen en un medio de dos dimensiones. Se expone el tema de recursividad y la refactorización es un tema central en la implementación del algoritmo.

Se explican y se ven temas relevantes como son la reflexión y refracción debido a que como siempre se está operando con la luz y con superficies con diferentes características. La incidencia de la luz, es un tema para dar un tratamiento adecuado. El tema de oversampling también está explicado dentro del ámbito de la computación gráfica.

Se tocan temas que resultan importantes para seguir avanzando en el campo de las imágenes en tercera dimensión, de la computación gráfica. Para poder hacer implementaciones con más componentes que le dan a las figuras nuevas cualidades. Hay temas que son importantes, estos se verán después de haber dominado los temas básicos y el alumno ya sea capaz de realizar un Raytracer y quiera dar mejor calidad a sus imágenes 3D. Los temas adicionales son: malla de triángulos, texturizado y procesamiento paralelo, Así como algunas otras estructuras y algoritmos que permiten optimizar el algoritmo y permite agregar nuevas cosas interesantes.

Recapitulando, las prácticas del Raytracer permitirán al estudiante de Ingeniería en Computación:

- Tener conocimientos básicos sólidos de computación gráfica.
- Tener conocimientos de los antecedes necesarios para la graficación.
- Obtener una herramienta didáctica de aprendizaje que incluye teoría y práctica.
- Adquirir conocimiento de computación gráfica, de manera gradual y progresiva.
- Aprender a seguir una metodología para programar proyectos grandes.
- Aprender a modularizar un proyecto mayor en pequeñas partes.
- Aprender a realizar pruebas como parte del desarrollo de programas.
- Saber implementar algoritmos computacionales.
- Aplicar conceptos de geometría analítica, algebra vectorial y física general.
- Implementar seudocódigo de forma modular.
- Aprender a escalar proyectos gracias a una metodología modular.
- Aprender a trazar rayos como parte de un método eficaz de creación de imágenes.
- Aprender a crear imágenes de una gran complejidad de la forma más eficiente.
- Entender la importancia de una escena virtual en la creación de imágenes.
- Aprender a crear imágenes que tengan texturas creando un realismo visual.
- Aprender a manejar elementos opacos y trasparentes, así como sus respectivas gamas.
- Manejar los conceptos de luces y sombras dentro del ámbito gráfico.
- Aprender los conceptos de reflexión y refracción para imágenes digitales.
- Aprender a optimizar el algoritmo Raytracing.
- Comprender conceptos de estructuras de datos para mejoras del proyecto.

2 CAPÍTULO II. IMPLEMENTACIÓN BÁSICA DE UN RAYTRACER

2.1 Práctica I Matemática vectorial para cómputo gráfico

Objetivo:

El alumno será capaz de programar las operaciones fundamentales del algebra vectorial: suma de vectores, resta de vectores, producto escalar, producto vectorial, módulo de un vector y normalización. Esto lo hará basado en la teoría contenida en esta práctica. Además, al contar con el seudocódigo de las operaciones vectoriales, podrá realizar el programa de una forma más sencilla.

Introducción:

Para poder implementar el algoritmo Raytracing, es necesario conocer el álgebra vectorial, de ahí la importancia de tener claros los conceptos básicos.

Algebra Vectorial. La parte de las matemáticas que se encarga de la manipulación y operación de los vectores. Los vectores tienen un sinfín de aplicaciones y se utilizan en distintos campos. Sólo se verá la parte que está relacionada con el espacio tridimensional y bidimensional, debido a que es donde las imágenes en 3D tienen lugar. Los objetos geométricos en un espacio de dos o tres dimensiones, son más fáciles de estudiar cuando se utilizan las cantidades vectoriales (vectores).

Vector. Es un ente que tiene muchos significados dependiendo del campo donde sea utilizado, aquí simplemente expondremos la definición matemática y su representación gráfica, lo cual es más que suficiente para nuestros objetivos. También se definirá que es un vector en el campo de las ciencias computacionales pero eso se realizará en la práctica 3.

Un vector es un objeto geométrico que tiene magnitud (longitud) y una dirección. Los vectores se pueden sumar y por lo tanto se pueden restar, ya que la resta es una particularidad de la suma. Se pueden realizar otras operaciones, dentro de las más importantes están el producto punto y el producto vectorial. Un vector es representado por un segmento de línea con una dirección definida. Un segmento dirigido que conecta un punto inicial A con un punto final B (Ver Figura 2.1.1)

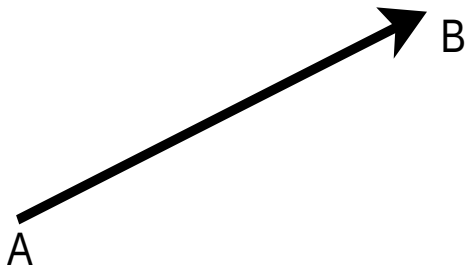


Figura 2.1.1 Vector

Un vector es lo que se necesita para "llevar" el punto A al punto B. La magnitud del vector es la distancia entre los dos puntos y la dirección se refiere a la dirección de desplazamiento de A a B. Esto se representa con una flecha. Las operaciones algebraicas de números reales tales como suma, resta, multiplicación, y la negación tienen análogos en las operaciones de vectores. Debido a esto las operaciones tienen también algunas propiedades como son: la propiedad conmutativa, asociativa y distributiva. Estas operaciones y leyes asociadas se deben a su carácter matemático que es estudiado de una forma más exhaustiva por el álgebra lineal. Pero para realizar las operaciones fundamentales simplemente con el entendimiento dentro del espacio (plano cartesiano) es más que suficiente.

En la física y en su aplicación práctica, la ingeniería, entes como la velocidad, aceleración, fuerza y muchas otras cantidades físicas, pueden ser descritos como vectores (magnitud y dirección). La representación geométrica y matemática de los vectores depende de un sistema coordenado donde serán inscritas.

El vector es una túpla de *n* números reales, en nuestro caso, dos o tres dimensiones.

Para su representación la letra que indica el vector va testada.

Dos dimensiones (*n*=2):

$$\bar{v} = (a_1, a_2)$$

Tres dimensiones (*n*=3):

$$\bar{v} = (a_1, a_2, a_3)$$

Escalar. Un escalar para las matemáticas y la física; por lo tanto también para la ingeniería, es una magnitud que no tiene componente vectorial, se representa por medio de números reales o complejos.

Sistema Coordenado. El sistema en el cual las coordenadas determinan la posición de un punto u otro elemento geométrico. El orden de las coordenadas es importante ya que cada una de las coordenadas está asociada a una referencia (eje).
Existen muchos sistemas coordenados. A continuación algunos ejemplos.

Recta numérica. El sistema coordenado más simple, con un simple número real se puede definir un punto en este sistema de una dimensión. Figura 2.1.2.

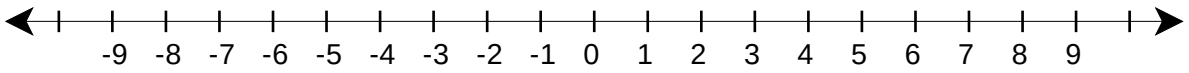


Figura 2.1.2 Recta numérica

Sistema cartesiano. Un espacio de dos o tres dimensiones en donde los puntos están definidos por túplas de números reales. El espacio de dos dimensiones, es el sistema cartesiano en el plano. Consta de dos ejes perpendiculares llamados coordenados, donde se intersectan se denomina el origen. Si bien con lo anterior es más que suficiente para definir un sistema cartesiano de dos dimensiones, se procura que los ejes tengan una dirección horizontal y vertical respectivamente. Esto se realiza por practicidad debido a que es más fácil trabajar así. El eje horizontal se le nombra de las abscisas (x) y al vertical de las ordenadas (y). Figura 2.1.3.

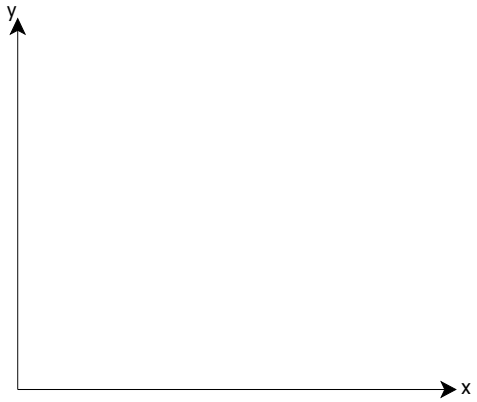


Figura 2.1.3 Plano cartesiano dos dimensiones

Un punto en el sistema coordenado de dos dimensiones (Figura 2.1.4) se define como la pareja de coordenadas:

$$P = (x, y)$$

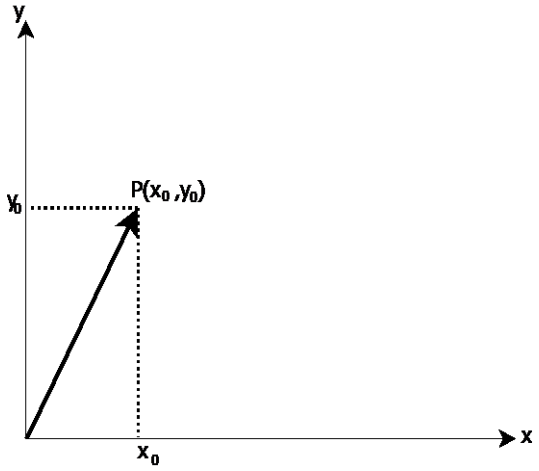


Figura 2.1.4 Punto en un sistema cartesiano de dos dimensiones

El caso del espacio de tres dimensiones está definido por tres ejes perpendiculares entre sí, cada uno perpendicular a los otros dos. Se cortan en un punto que es denominado origen. Los ejes coordenados se les nombra normalmente con las letras X,Y y Z. Los ejes X y Y se acostumbran dibujar en el plano horizontal y por lo tanto el eje Z, verticalmente. Figura 2.1.5.

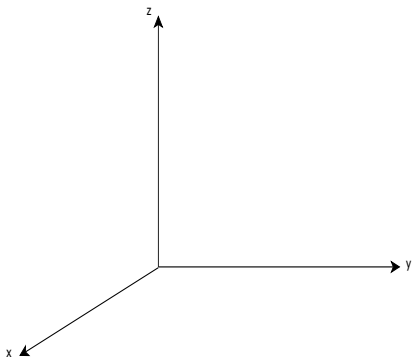


Figura 2.1.5 Sistema cartesiano de tres dimensiones

Un punto en el sistema coordenado de tres dimensiones Figura 2.1.6, se define como la terna de coordenadas:

$$P = (x, y, z)$$

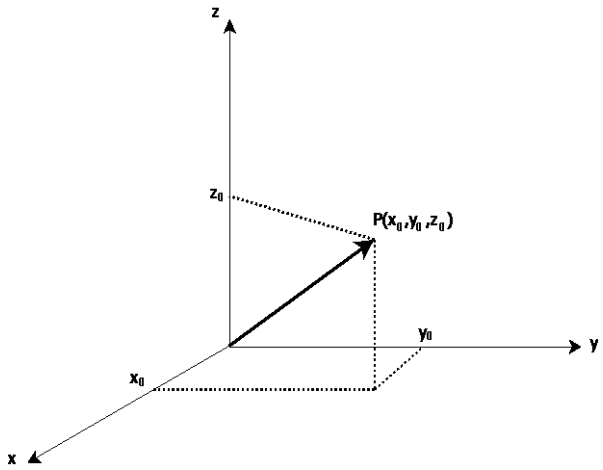


Figura 2.1.6 Punto en un sistema de tres dimensiones

Determinante. Es una notación matemática en forma de tabla cuadrada. Pueden existir de múltiples ordenes 1x1, 2x2, 3x3,…*nxn*, donde n es un número entero.

Determinante 2x2

$$|A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

Determinante 3x3

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

Determinante *n x n*

$$|A| = \begin{vmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{n1} & \dots & a_{nn} \end{vmatrix}$$

Determinantes de tercer orden. Este puede ser resuelto de la siguiente manera.

$$\begin{vmatrix} a & b & c \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2\, b_3 - a_3\, b_2)a \;+\; (a_3\, b_1 - a_1\, b_3)b \;+\; (a_1\, b_2 - a_2\, b_1)c$$

Vectores unitarios i,j,k. Un vector $\bar{a} = (a_1, a_2, a_{3n})$ se puede expresar en términos de los vectores unitarios i, j, k que tienen la dirección de los ejes coordenados y cuyo módulo es 1. Figura 2.1.7.

$$\bar{a} = a_1 i + a_2 j + a_3 k$$

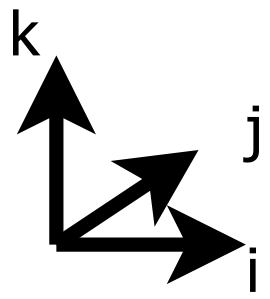


Figura 2.1.7 Vectores unitarios i, j, k

Ángulo entre dos vectores. Figura 2.1.8. Se define mediante la siguiente expresión:

$$\theta = ang\cos\frac{\bar{a}\cdot\bar{b}}{|\bar{a}||\bar{b}|}$$

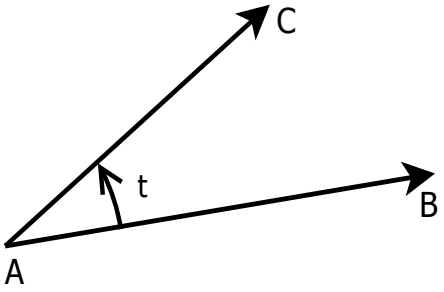


Figura 2.1.8 Ángulo entre vectores

Operaciones de vectores.

Suma de vectores. Dos vectores en el espacio de n dimensiones, $\bar{a} = (a_1, a_2, \dots, a_n)$ y $\bar{b} = (b_1, b_2, \dots, b_n)$, la suma es el vector que se obtiene sumando sus compontes correspondientes.

$$\bar{a} + \bar{b} = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$$

Resta de vectores. Dos vectores en el espacio de n dimensiones, $\bar{a} = (a_1, a_2, \dots, a_n)$ y $\bar{b} = (b_1, b_2, \dots, b_n)$, la resta es el vector que se obtiene restando sus compontes correspondientes.

$$\bar{a} - \bar{b} = (a_1 - b_1, a_2 - b_2, \dots, a_n - b_n)$$

Multiplicación por un escalar. Si β es un número real (escalar) y tenemos un vector $\bar{a} = (a_1, a_2, \dots, a_n)$ en el espacio de n dimensiones, el producto $\beta\bar{a}$ es el vector obtenido por la multiplicación de cada componente de $\bar{a}$ por β.

$$\beta\bar{a} = (\beta a_1, \beta a_2, \dots, \beta a_n)$$

Producto vectorial. Se necesitan dos vectores para realizar la operación. Sean $\bar{a} = a_1 i + a_2 j + a_3 k$ y $\bar{b} = b_1 i + b_2 j + b_3 k$ dos vectores en el espacio de tres dimensiones. Figura 2.1.9. El producto vectorial a cruz b está definido por el vector:

$$\bar{a}\, x\, \bar{b} = (a_2\, b_3 - a_3\, b_2) i + (a_3\, b_1 - a_1\, b_3) j + (a_1\, b_2 - a_2\, b_1) k$$

Otra representación es por medio del determinante de tercer orden:

$$\bar{a}\ x\ \bar{b}\ = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2\ b_3 - a_3\ b_2)i\ +\ (a_3\ b_1 - a_1\ b_3)j\ +\ (a_1\ b_2 - a_2\ b_1)k$$

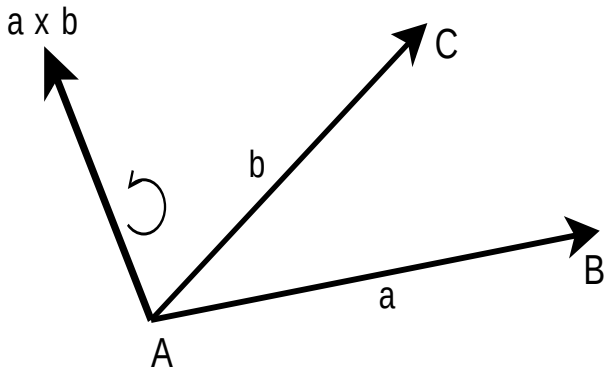


Figura 2.1.9 Producto vectorial

Producto escalar de dos vectores. Dos vectores en el espacio de n dimensiones, $\bar{a} = (a_1, a_2, \dots, a_n)$ y $\bar{b} = (b_1, b_2, \dots, b_n)$, $\bar{a}$ punto $\bar{b}$ es:

$$\bar{a} \cdot \bar{b} = \sum_{k=1}^{n} a_k b_k\ =\ a_1 b_1 +\ a_2 b_2 + \cdots + a_n b_n$$

El resultado del producto escalar de dos vectores es un escalar (número real).
Al producto escalar también se le conoce producto punto o producto interno.

Módulo de un vector. Es la magnitud del vector, se simboliza $|\bar{a}|$ al módulo del vector $\bar{a}$, el cual se obtiene de la raíz cuadrada de la suma de los cuadrados de cada uno de los componentes del vector. $|\bar{a}| = \sqrt[2]{a_1^2 + a_2^2 +, \dots, a_n^2}$ que como se puede observar es la aplicación del teorema de Pitágoras.
El modulo del vector también es llamado norma
Normalización y vectores unitarios. Sea un vector $\bar{a} = (a_1, a_2, a_3)$ el vector unitario es $\overline{a_u} = \left(\frac{a_1}{|\bar{a}|}, \frac{a_2}{|\bar{a}|}, \frac{a_3}{|\bar{a}|}\right)$

Desarrollo:

Después de entender toda la teoría, el alumno codificará el seudocódigo. Es preferible que utilice un lenguaje orientado a objetos, aunque también se puede utilizar un lenguaje de programación estructurada.

Las ventajas del lenguaje orientado a objetos es que se definirá un objeto (vector, luz, color, material etc.) cada objeto con sus respectivas funciones. Como en ocasiones se va a operar a nivel de objetos, de ahí la conveniencia de utilizar un lenguaje orientado a objetos. Si se utiliza un lenguaje estructurado tendrá que manejarlo todo por funciones.

Definiremos un objeto llamado vector que contiene las funciones: producto escalar, producto vectorial, módulo, suma de vectores y normalizar.

Pseudocódigo

Producto escalar. Producto escalar de dos vectores de tres dimensiones a y b
Las componentes de a son: a1, a2, a3, componentes de b: b1, b2, b3
Inicio
Res = a1*b1+ a2*b2+ a3*b3
Fin

Producto vectorial. Producto vectorial de dos vectores de tres dimensiones a y b
Las componentes de a son: a1, a2, a3, componentes de b: b1, b2, b3
Inicio
(r1,r2,r3) = (a2*b3 - a2*b2, a3*b1 - a1*b3, a1b2 - a2*b1)
Fin

Módulo. Módulo de un vector a de tres dimensiones.
Las componentes de a son: a1, a2, a3.
Inicio
Res = raíz cuadrada(a1*a1 + a2*a2 + a3*a3)
Fin

Suma de vectores. Suma de vectores de tres dimensiones a y b
Las componentes de a son: a1, a2, a3, componentes de b: b1, b2, b3
Inicio
(r1,r2,r3) = (a1 + b1, a2 + b2, a3 + b3)
Fin

Normalizar. Normalizar un vector a de tres dimensiones.
Las componentes de a son: a1, a2, a3.
Inicio
(r1,r2,r3) = (a1/raíz cuadrada(a1*a1 + a2*a2 + a3*a3), a2/raíz cuadrada(a1*a1 + a2*a2 + a3*a3), a2/raíz cuadrada(a1*a1 + a2*a2 + a3*a3))
Fin

También puede ser:

Inicio
(r1,r2,r3) = (a1/ResM, a2/ ResM, a2/ ResM))
Fin

Donde ResM es el resultado de la función Módulo.
Uniendo todas las partes en un seudocódigo global:

```
Inicio
Se crea el objeto Vector3
Se inicializa variables x, y, z
Método P Escalar = a1*b1+ a2*b2+ a3*b3
Método P Vectorial (r1,r2,r3) = (a2*b3 - a2*b2, a3*b1 - a1*b3, a1b2 - a2*b1)
Método de Módulo = raíz cuadrada(a1*a1 + a2*a2 + a3*a3)
Método de Suma (r1,r2,r3) = (a1 + b1,  a2 + b2,  a3 + b3)
Método Normalizar (r1,r2,r3) =
(a1/Método Módulo, a2/ Método Módulo, a2/ Método Módulo))
Fin
```

Pruebas del código.

La prueba sería:

- el módulo de un vector de tres dimensiones,
- el producto escalar de dos vectores de tres dimensiones,
- el producto vectorial de dos vectores de tres dimensiones
- la suma de dos vectores de tres dimensiones
- la normalización de un vector de tres dimensiones

Se comienza a codificar con base en el pseudocódigo.
Una vez que el alumno haya codificado el pseudocódigo, probará que su programa funciona. Escribirá un programa que le permita probar el programa de la práctica I.

Tabla de pruebas

Caso	**Entrada**	**Salida**
Módulo de un vector	v1 =vector(1,2,3)	3.741657386
Producto escalar de dos vectores	v1 =vector(1,2,3) v2=vector(1,1,1)	6
Producto vectorial de dos vectores	v1 =vector(1,2,3) v2=vector(1,1,1)	(-2,2,-1)
Suma de dos vectores	v1 =vector(1,2,3) v2=vector(1,1,1)	(2,3,4)
Normalización de un vector	v2=vector(1,1,1)	(0.801924, 0.534530, 0.267265)

Tabla 2.1.1 Pruebas

Resultados: *(El alumno colocará aquí cuales han sido los resultados de su práctica). Ejemplo: código fuente exclusivamente del pseudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto I en la práctica V.*

Conclusiones: *(El alumno colocará aquí sus conclusiones por realizar la práctica). Ejemplo: Se aprendieron los conceptos fundamentales de algebra vectorial la cual son la base para poder desarrollar un raytracer, ya que son operaciones que el programa tiene que hacer constantemente para generar las imágenes 3D…*

Bibliografía:

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995

CASTAÑEDA De I. P., Érik, Geometría analítica en el espacio; México; Facultad de Ingeniería – UNAM 2003.

2.2 Práctica II Fundamentos de color e Imágenes digitales

Objetivo:

El alumno será capaz de entender los conceptos de color así como saber como está constituida una imagen digital. Entendiendo estos conceptos y siguiendo el pseudocódigo será capaz de generar un archivo ppm, además de codificar el programa que define el color en el Raytracer.

Introducción:

El color es un tópico importante para las imágenes digitales, los colores dentro de la computación gráfica funcionan distinto a como nosotros definimos los colores en el mundo físico, de ahí la importancia de conocer los aspectos fundamentales.

Imagen digital. Se define como una representación numérica, normalmente binaria, de una imagen de dos dimensiones. Dependiendo de la resolución de la imagen, puede ser una imagen vectorial o un mapa de bits.

El archivo de imagen contiene la información de los atributos de la imagen entre los cuales se encuentran dimensiones, tipo de codificación, etc. Además de los datos de la imagen en sí misma.

Mapa de Bits. Es una imagen creada sobre cuadrícula, que se guarda como un archivo. Cada uno de los elementos de la cuadricula se denomina píxel, el cual guarda la información del color. Cuantos más píxeles, mayor es la calidad de la imagen. Además hay píxeles que guardan más información que otros, dependiendo de la información del archivo. Las imágenes en mapa de bits están definidos por su altura y ancho, estás dimensiones son definidas en píxeles. También es relevante la profundidad de color (en bits por píxel), que determina el número de colores distintos que se pueden almacenar en cada punto individual. Factor importante para la calidad del color de la imagen digital.

La manipulación de una imagen en mapa de bits sin perder calidad es prácticamente imposible debido a que cada elemento de la matriz, forma a la imagen, y si se quiere agregar tamaño, la matriz se amplía dejando huecos que serán percibidos como falta de resolución.

Las imágenes en mapa de bits son más prácticas para tomar fotografías o filmar escenas,

Imágenes vectoriales (gráficos vectoriales). Están formadas por objetos geométricos independientes como esferas, triángulos, líneas, polígonos, y demás figuras. Estos elementos son definidos por medio de expresiones matemáticas. Lo cual puede crear imágenes con mayor calidad.
Los gráficos vectoriales pueden adaptar su resolución fácilmente a la de cualquier dispositivo de visualización. Las imágenes vectoriales se utilizan sobre todo para la representación de figuras geométricas con parámetros definidos.

Vector. En ciencias computacionales un vector se define como un arreglo de una dimensión

Píxel. Es la mínima unidad homogénea de una imagen digital, son los puntos de color de una imagen. En los monitores de computadoras se sigue el modelo de color RGB. Cada pixel está conformado por tres números que representan los colores primarios: el rojo, el verde y el azul, y con estos se pueden obtener los demás colores. La imagen digital al ser una matriz con una cantidad de elementos, estos son llamados pixeles que pueden tomar sólo un posible valor.

Vóxel. Es una palabra compuesta a partir de volumetric pixel (píxel volumétrico) que representa el concepto de la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional. Para crear una imagen en tres dimensiones, los vóxeles deben de tener el componente de la opacidad. Esto hace que se tenga un efecto en tres dimensiones.

Color. Modelo RBG de las siglas en inglés Red(Rojo), Green(verde), Blue(Azul) debido a que el color es creado en la mente del ser humano por medio de mecanismos mentales y fisiológicos del ojo humano, conociendo estas características se puede presentar una matriz con una configuración de colores básicos que cree la percepción de color en el ojo. Estás combinaciones se hacen con el rojo, verde y azul. Para obtener el color negro se tienen que apagar los tres colores (esto sería Rojo=0; Verde=0; Azul=0;). Si se quiere un color blanco los colores deben estar encendidos hasta el máximo color, por ejemplo si el máximo color es 255 entonces los tres colores deben de tener este mismo valor

(Rojo=255; Verde=255; Azul=255;). La manera gráfica se puede observar en la figura 2.2.2 Modelo RGB.

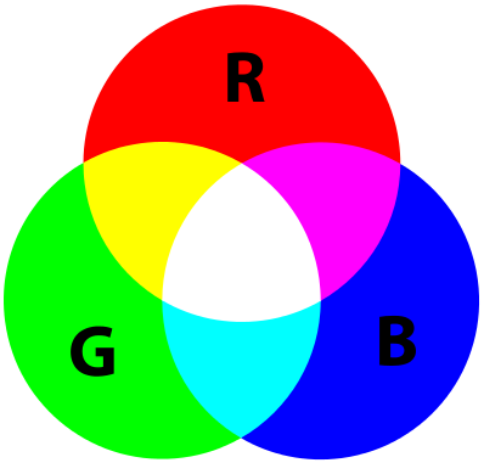


Figura 2.2.1 Modelo RGB[14]

Color en una pantalla de computadora. El color es formado por los píxeles. Cada píxel tiene interiormente 3 subpíxeles, uno rojo, uno verde y otro azul; dependiendo de que subpíxel este encendido, será el valor que tomará el píxel. Siguiendo la composición de colores RGB.

La manera de organizar los subpíxeles de un monitor es variable entre los dispositivos. Se suelen organizar en líneas verticales, algunos los organizan en puntos formando triángulos. Para mejorar la sensación de movimiento. La mejor configuración es en diagonal o en triángulos. El conocimiento del tipo de organización de píxeles, puede ser utilizado para mejorar la visualización de imágenes de mapas de bit usando renderizado de subpíxeles.

La mayor parte de los monitores tienen una profundidad de 8 bits por color (24 bits en total), es decir, pueden representar aproximadamente 16,8 millones de colores distintos.

La profundidad de color o bits por pixel (bpp). Es la cantidad de bits de información necesarios para representar el color de un píxel en una imagen digital. Porque se usa el

[14] *Public Domain* (Dominio Público)

sistema binario de numeración, una profundidad de bits de n implica que cada píxel de la imagen puede tener 2n posibles valores y por lo tanto, representar 2n colores distintos.

Los arreglos de 8 bits son las unidades básicas de información en los dispositivos de almacenamiento, los valores de profundidad de color son divisores o múltiplos de 1, 2, 4, 8, 16, 24 y 32, con la excepción de la profundidad de color de 10 o 15, usada por ciertos dispositivos gráficos.

Color verdadero. Los gráficos de color verdadero utilizan un método de almacenamiento de la información de la imagen donde cada píxel está representado por 3 o más bytes. Los bits están divididos en valores para el componente rojo, el verde y el azul (RGB, del inglés red, green, blue) del color final.

Cada uno de las componentes RGB disponen de 8 bits,2^8, 256 valores de cada color. Un total de 16 777 216 posibles colores para cada píxel. La razón de que se denomine color verdadero es debido a que es aproximadamente el número de colores que el ojo humano puede detectar.

Formatos de archivo de imagen. Es la forma de organizar y almacenar imágenes digitales. Los archivos de imagen se componen de datos digitales (bytes). Los formatos pueden ser decodificados para presentarse en la pantalla de la computadora. Un formato de archivo de imagen almacena datos en formatos sin comprimir, comprimido, o vectoriales. Una vez decodificada la imagen, ésta se convierte en una matriz de píxeles, cada uno de los cuales tiene un número de bits para designar su color.

El tamaño del archivo está relacionado con la cantidad de bits de la imagen, a mayor cantidad de bits por imagen el archivo será más pesado. Existen métodos de compresión de imágenes, estos algoritmos hacen que la imagen conserve las cualidades reduciendo el tamaño del archivo. Existen múltiples formatos como JPEG, TIFF, GIF, BMP, PNG, PPM y muchos más.

PPM. Portable PixMap. Estos archivos son ASCII puros o archivos binarios sin el formato ASCII. Tienen un encabezado con los datos sobre la funcionalidad básica, lo cual sirve para la conversión a mapa de píxeles. PPM es un formato gráfico simple en color. Utiliza 24 bits por píxel: 8 para el rojo, 8 para el verde y 8 para el azul si el valor máximo de gris es de 255, utiliza 16 bits para el rojo, 16 para el verde y 16 para el azul si el valor es mayor que 255 y menor que 65536. Un archivo PPM contiene texto plano y puede ser modificado con un simple procesador de texto, también existe la versión binaria no legible por procesadores de texto normalmente. Está relacionado con los formatos PGM (escala de grises) y PBM (blanco y negro). Los formatos Netpbm son unos formatos de imagen sin compresión, diseñados para ser extremadamente fáciles de comprender por humanos y por computadoras.

Todos los formatos Netpbm comparten una estructura similar:

- Número mágico
- Ancho Alto
- Máximo color
- Píxeles

Todos los formatos de dividen en 2 subformatos: ASCII y binario. El formato ASCII es legible por humanos y computadoras, los números se representan en decimal. El formato binario es legible por computadoras, es más compacto que el anterior.

En resumen el archivo PPM es una larga cadena de caracteres (números), los números van generalmente del 0 al 255, cada número cuando sea interpretado por la computadora será el valor del píxel en el mapa de bits.

Desarrollo:

Después de entender toda la teoría, el alumno codificará el pseudocódigo. Es preferible que utilice un lenguaje orientado a objetos, se puede utilizar un lenguaje de programación estructurada.

Las ventajas del lenguaje orientado a objetos es que se definirá un objeto (vector, luz, color, material etc.) cada objeto con sus respectivas funciones. Como en ocasiones se va

a operar a nivel de objetos, de ahí la conveniencia de utilizar un lenguaje orientado a objetos. Si se utiliza un lenguaje estructurado tendrá que manejarlo todo por funciones.

Pseudocódigo

Se van a definir tres objetos color, imagen y material, esto permitirá crear una imagen. El programa como resultado arrojará un archivo PPM.

```
Inicio

    Inicio color

    Creamos un objeto Color que incluye las componentes RGB con rango [0.0, 1.0]
    después se transforma al rango [0, 255] 8 bits por componente, las componentes
    son de sólo lectura.

    Definir como color predeterminado al color negro (r = 0.0, g = 0.0, b = 0.0)

    Definir el operador overloading que se encarga de sumar y multiplicar

    Función overloading {

    Suma (R + componente del R, G + componente del G, B + componente del B)

    Nuevo color = (R * f, G * f, B * f) donde f es un escalar

                      }

    Convertir  (Val1, Val2, Val3) = (Max (0.0, MIN(R*255.0,0.0), Max (0.0,
    Min(G*255.0,0.0), Max (0.0, Min(B*255.0,0.0))

    Fin color

    Inicio imagen

    Creamos el objeto Imagen, que contendrá los datos de todos los pixeles
    (colores) de nuestra imagen.

    Definimos el ancho width y la altura height, cada campo de la matriz será un
    objeto de color.

    Guardar los valores de width y height

    Crear método que recibe una ruta/nombre de archivo y crea una imagen en formato
    PPM en la ruta indicada.

    Escribir en el archivo, P3 componentes por pixel, width y height, 255 valores
    posibles

    Escribir cada renglón de valores consecutivamente separados por espacios y en
    el momento de llegar al límite del ancho, hacer un salto de línea.

    Fin Imagen
```

```
Inicio material

Crear el objeto Material

Definimos el componente color

Inicializamos color en 0.0

Llamar los valores del objeto color

Color del componente color = color del objeto color

            color = color

También se definen los componentes difuso, especular, brillo, reflexión,
transmitividad, IOR (índice de refracción)

           diffuse = diffuse
           specular = specular
           shininess = shininess
           reflect = reflect
           transmit = transmit
            ior = ior

Fin material


Fin Total
```

Nota: el objeto material almacenará por medio de funciones todas las propiedades: color, diffuse, specular, shininess, reflect, transmit, ior. Pero en esta práctica sólo guardará color, en prácticas posteriores se agregaran las demás funciones.

Pruebas

Caso	**Entrada**	**Salida**
Generación de un archivo ppm de una imagen	NA	test.ppm

Tabla 2.2.1 Pruebas

Nota: El archivo PPM si se abre con un editor de texto sólo mostrarán una matriz de números, por lo cual es necesario un visor de imágenes PPM. El alumno puede escoger el de su preferencia. La recomendación sería *FastStone Image Viewer.*

Resultados: *(el alumno colocará aquí cuales han sido los resultados de su práctica). Ejemplo: código fuente exclusivamente del seudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto I en la práctica V.*
Las impresiones de pantalla de los resultados obtenidos.
Cualquier otra evidencia.
Conclusiones: *(el alumno colocará aquí sus conclusiones por realizar la práctica). Ejemplo: Se aprendieron los conceptos fundamentales de que es el color y que es una imagen dentro de la computación gráfica. Lo cual es muy importante para poder construir un raytracer…*

Bibliografía:

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Kevin Suffern; Ray Tracing from the Ground Up; A K Peters/CRC Press (September 6, 2007)

2.3 Práctica III
Definición de primitivas geométricas y objetos

Objetivo

El alumno entenderá el concepto de primitiva geométrica y ocupará el plano y la esfera desde el punto de vista de la geometría analítica en el desarrollo de la práctica para presentar las primeras imágenes del Raytracer utilizando el lenguaje de programación de su interés.

Introducción

Primitivas geométricas

Son formas geométricas consideradas así por su constitución básica en las partes que la conforman, esto es, formadas por puntos, rectas y planos. Se conocen también con el nombre de primitivas geométricas al círculo, el triángulo y el cuadrado. En cuanto a primitivas tridimensionales existen los cilindros, el tubo, la esfera, el cubo, entre otros.

El plano

En geometría, un plano es un objeto que tiene ancho y largo, sin altura ni grosor. Se puede pensar como un conjunto de puntos infinitos en dos dimensiones.

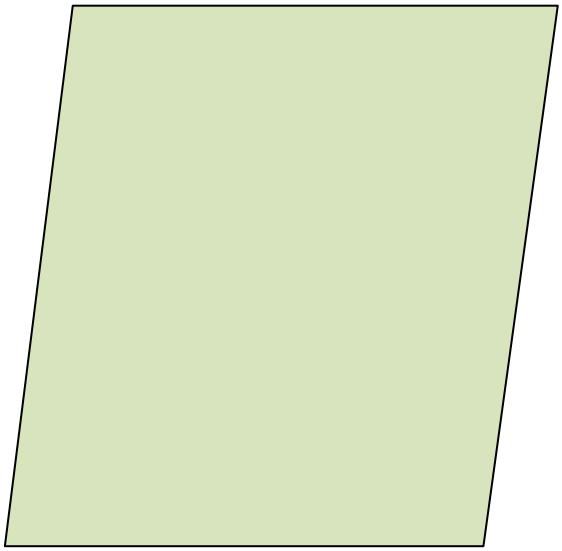


Fig. 2.3.1 Plano

Ecuación general del plano

Cualquier plano se puede expresar como una ecuación del plano de la primera forma:

$$A_x + B_y + C_z + D = 0$$

Donde A, B y C no pueden ser 0 al mismo tiempo.

Ecuación vectorial del plano

También se puede representar como se muestra a continuación:

$$(x, y, z) = (x_1, y_1, z_1) + k(u_x, u_y, u_z) + t(v_x, v_y, v_z)$$

Ecuación paramétrica del plano

A partir de la ecuación vectorial del plano, si realizamos operaciones en el 2º término de la igualdad:

$$(x, y, z) = (x_1 + ku_x + tv_x, y_1 + ku_y + tv_y, z_1 + ku_z + tv_z)$$

El valor de *x* depende de: x_1, ku_x, tv_x

El valor de *y* depende de: y_1, ku_y, tv_y

El valor de *z* depende de: z_1, ku_z, tv_z

Finalmente:

$$x = x_1 + ku_x + tv_x$$
$$y = y_1 + ku_y + tv_y$$
$$z = z_1 + ku_z + tv_z$$

Esta es la ecuación del plano en su forma paramétrica.

Ecuación del plano en forma implícita

Pasamos a la izquierda del signo igual cada uno de los valores que se encuentran a la derecha de dicho signo: $x - x_1 - ku_x - tv_x = 0$

$$y - y_1 - ku_y - tv_y = 0$$
$$z - z_1 - ku_z - tv_z = 0$$

Ecuación del plano en segmentos

Si el plano cruza los ejes OX, OY y OZ en los puntos con coordenadas (a, 0, 0), (0, b, 0) y (0, 0, c), entonces puede calcularse, utilizando la fórmula de ecuación del plano en segmentos:

$$\frac{x}{a} + \frac{y}{b} + \frac{z}{c} = 1$$

Ecuación del plano, que pasa por un punto, perpendicularmente al vector normal

Para formular la ecuación del plano, sabiendo las coordenadas del punto del plano $M(x_0, y_0, z_0)$ y vector normal del plano n = {A; B; C} se puede utilizar la fórmula siguiente:

$$A(x - x_0) + B(y - y_0) + C(z - z_0) = 0$$

Ecuación del plano que pasa por tres puntos dados, que no están en una recta

Si hay coordenadas dadas de tres puntos $A(x_1, y_1, z_1)$, $B(x_2, y_2, z_2)$ y $C(x_3, y_3, z_3)$, que están en un plano, entonces la ecuación del plano se puede calcular por la fórmula siguiente:

$$\begin{vmatrix} x - x_1 & y - y_1 & z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix} = 0$$

La esfera

Una esfera es una superficie originada por una circunferencia que gira sobre su diámetro.

Los elementos identificados propios de una esfera son:

Centro: Punto interior que equidista de cualquier punto de la superficie de la esfera.

Radio: Distancia del centro a un punto de la superficie de la esfera.

Cuerda: Segmento que une dos puntos de la superficie esférica.

Diámetro: Cuerda que pasa por el centro.

Polos: Son los puntos del eje de giro que quedan sobre la superficie esférica.

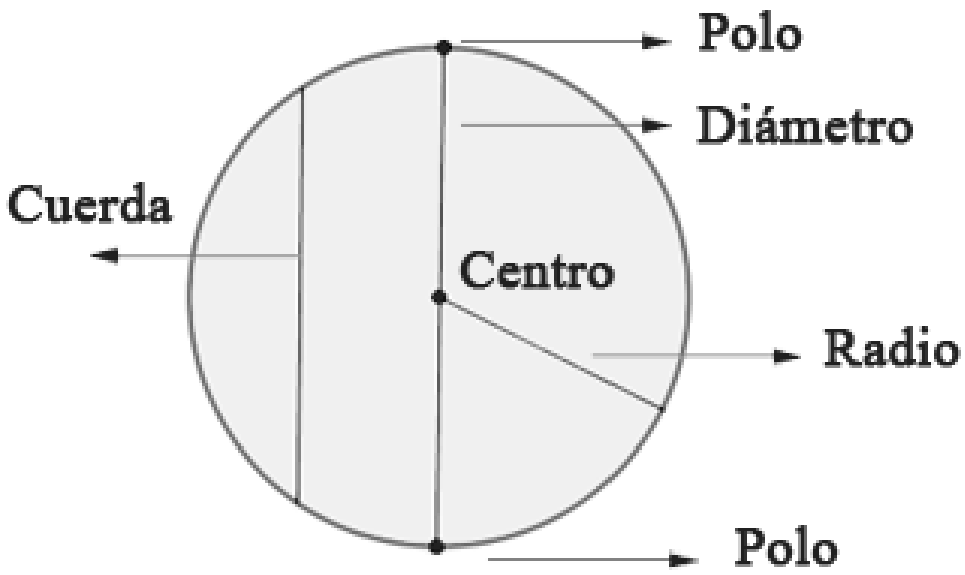


Fig. 2.3.2 Elementos de una esfera[15]

FÓRMULAS DE LA ESFERA	
Radio de una esfera	$R = \sqrt{d^2 + r^2}$
Área de la superficie esférica	$A = 4 \cdot \pi \cdot r^2$
Volumen de la esfera	$V = \frac{4}{3}\pi \cdot r^3$

Tabla 2.3.1 Fórmulas

[15] http://www.vitutor.net/2/2/33.html

Ecuación reducida de la esfera

Cuando el centro es el origen de coordenadas la ecuación que deben satisfacer los puntos X(x,y,z) para pertenecer a la esfera es: $x^2 + y^2 + z^2 = r^2$. Se le llama ecuación reducida.

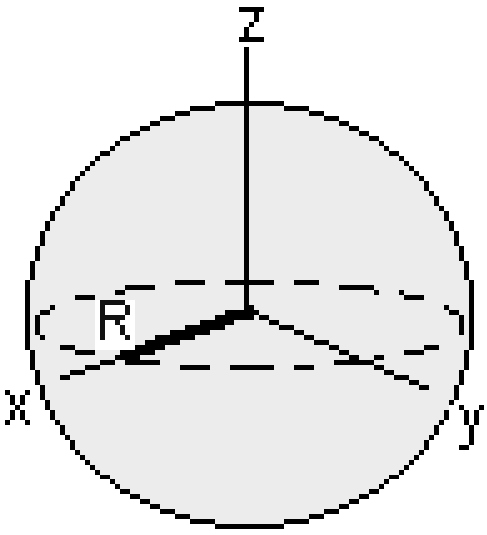


Fig. 2.3.3 Esfera con centro en el origen[16]

Si el centro fuese el punto C(a,b,c) la ecuación sería:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$$

[16] http://www.ehu.eus/juancarlos.gorostizaga/apoyo/geometr2.htm

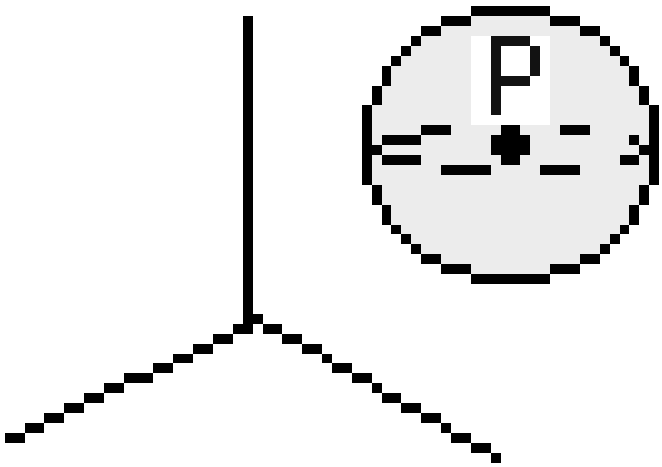


Fig. 2.3.4 Esfera con centro en cualquier otro punto fuera del origen[17]

Plano tangente a la esfera

El plano que toca a la esfera en un solo punto es llamado plano tangente. Cada punto de la esfera tiene asociado un plano tangente.

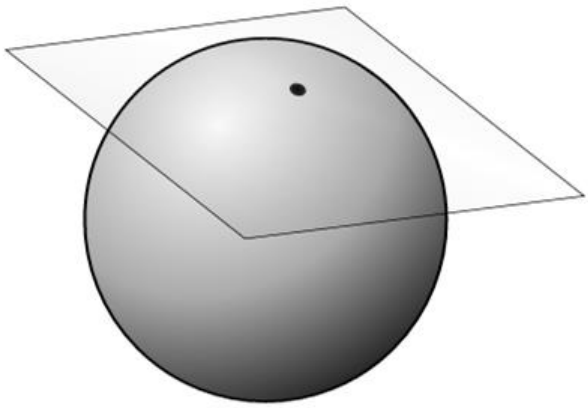


Fig. 2.3.5 Plano tangente a la esfera[18]

[17] http://www.ehu.eus/juancarlos.gorostizaga/apoyo/geometr2.htm
[18] http://www.ehu.eus/juancarlos.gorostizaga/apoyo/geometr2.htm

Desarrollo

En la práctica anterior se reforzaron los conocimientos referentes a operaciones con vectores y que son básicos para entender cómo trabaja el método de Raytracer utilizado en cómputo gráfico. Ahora, en la introducción de esta práctica, se habló de primitivas geométricas y entonces se iniciará mencionando que se ocupará la esfera.

Se definirá una clase genérica de objeto de la cual se heredará tanto la primitiva geométrica a utilizar como los objetos virtuales.

La conveniencia de utilizar programación orientada a objetos, es que ya existen librerías precargadas que nos facilitan la definición de ciertos objetos como las primitivas geométricas que ocuparemos. La esfera como primitiva tiene una resolución infinita lo que significa que tiene infinitas caras, para definirla sólo requerimos de la posición o centro de la esfera y de su radio.

La segunda primitiva geométrica que se ocupará es un plano. A partir de la definición de este concepto que se repasó en la introducción de esta práctica, se debe considerar un plano infinito. Lo único que se requiere para definirlo es su normal, y la distancia a la que el plano está del origen.

$$A(x - x_0) + B(y - y_0) + C(z - z_0) = 0$$

Pseudocódigo

definir objeto3D
Todos los objetos tienen un tipo ("plano", "esfera", etc...) y un material que corresponderá a una referencia en la lista de materiales.

definir variables tipo, material
definir esfera

definir variable centro, radio

Así como su material, también se necesita definir su centro (vector) y su radio (escalar)

definir variables iniciales: material, centro, radio

esfera, material

variable1 = center

variable2 = radius

Ahora se calculará la normal en un punto de la esfera. (Dicho cálculo se ha revisado anteriormente en la Práctica 1.)

Un punto que pertenece al radio de la esfera es: (r_1,r_2,r_3)

Para calcular la normal en un punto de la esfera:

(r_1,r_2,r_3) = (a_1/Método Módulo, a_2/ Método Módulo, a_3/ Método Módulo))

El resultado corresponde al vector que va desde el centro hasta el punto, normalizado.

Plano

Al igual que la primitiva anterior, la primitiva geométrica plano requiere que sean definidos sus variables material, su normal y su distancia.

definir variables plano (material, normal, distancia)

plano, material

variable3 = normal

variable4 = distancia

Tomando como referencia la algebra vectorial respecto al plano, la normal de un plano en cualquier punto de este siempre es la normal del plano, así que simplemente devolvemos el vector almacenado.

obtener normal

devuelve normal

Pseudocódigo final

```
definir objeto3D
  definir variables tipo, material

definir valor inicial (tipo, material)
    variabletipo = tipo
    variablematerial = material
  fin

definir esfera
   definir variable centro, radio

definir variables iniciales: material, centro, radio
    esfera, material
    variable1 = center
    variable2 = radius
  fin

definir objeto3D
  definir variables normal, distancia

definir variables plano (material, normal, distancia)
    plano, material
    variable3 = normal
    variable4 = distancia
```

Pruebas

Caso	Entrada	Salida
Probar que los objetos esfera y plano se forman	NA	Datos de la esfera y plano.

Tabla 2.3.2 Pruebas

Resultados

El alumno describirá los resultados obtenidos del desarrollo de su práctica.

Conclusiones

El alumno plasmará sus comentarios que demostrarán que se reafirma el conocimiento adquirido.

Bibliografía

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Kevin Suffern; Ray Tracing from the Ground Up; A K Peters/CRC Press (September 6, 2007)

2.4 Práctica IV DEFINICION Y COMPOSICION DE UNA ESCENA VIRTUAL

Objetivo

Para esta práctica se enseñará la definición de la escena mediante la carga de un archivo de texto, en esta se cargará la escena y se guardaran todos los objetos, luces, materiales, imagen, etc.

Introducción

Tenemos que definir tres diferentes tipos de cosas en el espacio 3D: una cámara de la que se lanzan los rayos en la escena, objetos que pueden ser afectados por los rayos y se dibujan en la escena, y las luces que cambian el color de los rayos, por extensión, objetos colorantes, esto se puede observar en la Figura 2.4.1.

En este caso, se define estos objetos como objetos simples con vectores definidos como {x, y, z} objetos.

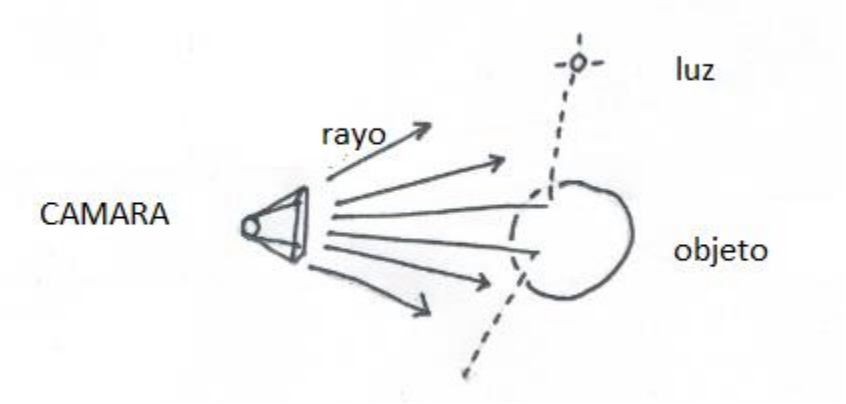


Figura 2.4.1 Introducción de escena[19]

[19] http://www.macwright.org/literate-raytracer/

Raytracer renderizar las escenas que contienen los objetos geométricos, luces, una cámara, un plano de la vista, un trazador y un color de fondo. En el Raytracer descrito en esta práctica, estos objetos se almacenan en un objeto del mundo. Por ahora, el mundo sólo almacenará los objetos y vista en planta. Las ubicaciones y orientaciones de todos los elementos de la escena se especifican en coordenadas mundo, que es un sistema de coordenadas cartesianas 3D.

Las coordenadas mundiales se conocen como coordenadas absolutas debido a que su origen y la orientación no están definidos este no es un problema. La única tarea del Raytracer es calcular el color de cada píxel, y los píxeles también se definen en coordenadas mundo.

Al crear cualquier tipo de gráficos por computadora, debe tener una lista de objetos que desea que el software renderize. Estos objetos son parte de una escena o mundo, Figura 2.4.2, por lo que cuando hablamos de “ver el mundo, " nos estamos planteando un desafío filosófico, pero sólo en referencia al raytracer dibujando los objetos de un determinado punto de vista. En los gráficos, este punto de vista se denomina el ojo o la cámara. A raíz de esta analogía, al igual que una cámara necesita película sobre la que se proyecta y graba la escena, en los gráficos tenemos una ventana de vista en el que dibujamos la escena. La diferencia es que mientras que en las cámaras la película se coloca detrás de la abertura o punto focal, en gráficos la ventana de vista está enfrente del punto focal. Así que el color de cada punto en la película real está causado por un rayo de luz (en realidad, un grupo de rayos) que pasa a través de la abertura y le pega a la película, mientras que en los gráficos por computadora cada píxel de la imagen final es causado por un rayo de luz simulada que golpea la ventana de vista en su camino hacia el ojo.

El objetivo es encontrar el color de cada punto de la ventana de visualización. Subdividimos la ventana de vista en pequeños cuadrados, donde cada cuadrado corresponde a un píxel en la imagen final. Si se desea crear una imagen con una resolución de 640x400, la ventana de vista es una rejilla de 640 cuadrados de ancho y 400 cuadrados de largo. El problema real, entonces, es la asignación de un color a cada cuadrado. Esto es lo que hace el trazado de rayos.

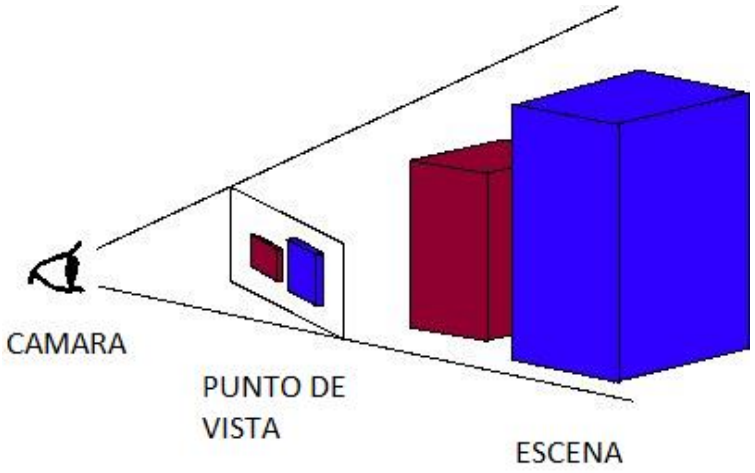


Figura 2.4.2 Ojo, ventana y escena[20]

Renderizado

El renderizado es el proceso de generar una imagen de 2D o 3D modelo (o modelos en lo que podría llamarse colectivamente un archivo de escena) por medio de programas de computadora. Además, los resultados de un modelo de este tipo pueden ser llamados a una representación. Un archivo de escena contiene objetos en una estructura del lenguaje o los datos estrictamente definidos como geometría, punto de vista, la textura, la iluminación, y la información de sombreado como una descripción de la escena virtual. Los datos contenidos en el archivo de escena se hacen pasar luego a un programa de renderizado para ser procesado y con salida a un archivo de imagen o imagen de mapa de bits gráficos digitales. El término "representación" puede ser por analogía una "representación del artista" de una escena. Aunque los detalles técnicos de los métodos de representación varían, los retos generales para superar en la producción de una imagen 2D a partir de una representación 3D almacenada en un archivo de escena se perfilan como el canal de gráficos a lo largo de un dispositivo de presentación, tal como

[20] https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

una GPU. Un GPU es un dispositivo especialmente diseñado capaz de ayudar a una CPU en la realización de cálculos complejos de representación. Si una escena es requerida renderear de manera realista y en condiciones de iluminación virtual, el software de procesamiento debe resolver la ecuación de la representación. La ecuación de la representación no da cuenta de todos los fenómenos de iluminación, pero es un modelo de iluminación general para las imágenes generadas por computadora. 'Renderizado' también se utiliza para describir el proceso de cálculo de efectos en un programa de edición de vídeo para producir la salida de vídeo final.

Cámara

Lo que sigue es la simplificación extrema de lo que existe en nuestro ojo o en una cámara. Una cámara simplificada está hecha de un centro óptico, eso significa que es el punto en el que todos los rayos de luz convergen, y un plano de proyección. Este plano de proyección se puede colocar en cualquier lugar, incluso dentro de la escena, ya que pasa a ser una simulación en una computadora y no una verdadera cámara (en el caso de la verdadera cámara, el plano de proyección estaría en la película o CCD). También contenía un sistema óptico que está ahí para hacer converger los rayos que pasan a través del diafragma. Una cosa importante de entender para el modelo físico de las cámaras es que tienen que dejar pasar un haz de luz, lo suficientemente grande como para que la película o la retina se inscriba. El sistema óptico debe estar configurado de modo que el punto de interés de la escena sea enfocado en el plano de proyección. A menudo se excluyen mutuamente, eso significa que no puede haber dos puntos, uno a lo largo y cerca de uno que puede ser a la vez fuerte al mismo tiempo. En la fotografía llamamos a esto la profundidad de campo. Nuestro modelo simplificado de la cámara nos permite simular el mismo efecto para una mejora de foto-realismo. Pero la principal ventaja que hemos comparado con el fotógrafo de la vida real es que podemos hacer esto fácilmente sin la restricción del sistema físico.

El desenfoque causado por la profundidad de campo es un poco exhaustivo para simular debido al trabajo añadido. La razón es que en nuestro caso vamos a tratar de simular este efecto con un método estocástico. Eso significa que se enviará una gran cantidad de rayos pseudo-aleatorios para tratar de dar cabida a la complejidad del problema. La idea detrás de esta técnica consiste en imaginar que disparar un rayo de fotones a través del sistema óptico y rayos que serán ligeramente desviados del centro óptico debido a la apertura del diafragma. Cuanto más grande es el agujero en el diafragma, más grande puede ser la desviación. Sin embargo, para simplificar, se considerará el sistema óptico como un cuadro negro. Algunas personas lograron con éxito representar imágenes con representaciones exactas de las lentes, etc. Lo único que importa a partir de ahora es a qué distancia de nuestro centro óptico están los puntos en la escena que se verán tan marcados, y desde qué dirección el fotón estaba golpeando nuestro plano de proyección de modo que se pueda seguir su camino inverso.

Es necesario utilizar las propiedades geométricas de los sistemas ópticos, estos se llaman los invariantes. En primer lugar cualquier rayo que pasa a través del propio centro óptico no se desvía de su curso. En segundo lugar, dos rayos que cruzan el plano "fuerte" en el mismo punto convergen en el mismo punto en el plano de proyección, esto es, por definición del plano "fuerte". Estas dos propiedades nos permitirán determinar el camino inverso del 100% de los rayos que golpean nuestro plano de proyección. Vea Figura 2.4.3.

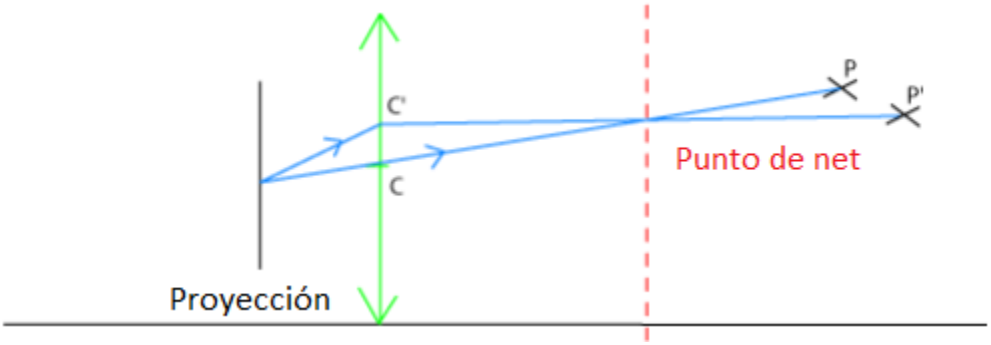


Figura 2.4.3 Profundidad del campo[21]

[21] https://www.ics.uci.edu/~gopi/CS211B

Campo de visión (Field of View - FOV)

El campo de visión es la medida del mundo observable que se ve en cualquier momento dado. El campo de visión se da generalmente como un ángulo para el componente horizontal o vertical del campo de visión. Un ángulo mayor indica un mayor campo de visión, sin embargo, dependiendo del método de escala FOV utilizado por el sistema, es posible que sólo afecta a la horizontal o la componente vertical del campo de vista.

Desarrollo

El proceso de creación de una imagen se iniciará con la construcción de los rayos que se han mencionado durante toda la tesis que se llamarán rayos primarios o rayos de cámara (Primero, debido a que estos son los rayos primeros se va a disparar en la escena. Los rayos secundarios son sombras rayos, por ejemplo, de lo que se hablará más adelante). ¿Qué se sabe acerca de estos rayos que ayudará a construirlos? Se sabe que empiezan desde el origen de la cámara. En casi todas las aplicaciones 3D, la posición por defecto de la cámara cuando se crea es el origen del mundo, que se define por el punto de coordenadas (0, 0, 0).

La película de cámaras oscuras del mundo real se encuentra detrás de la abertura, lo que hace que los rayos de luz por construcción geométrica que forma una imagen invertida de la escena. Sin embargo, esta inversión se puede evitar si el plano de la película se encuentra en el mismo lado que la escena (en frente de la abertura en lugar de detrás). Por convención, en el trazado de rayos, a menudo se coloca exactamente 1 unidad de distancia desde el origen de la cámara, pero en este caso, en este manual se coloca tanto la posición de la cámara como la posición de la cuadrícula, y con esto se calculará dicha distancia.

La tarea consiste en crear un rayo primario para cada pixel de la trama. Esto puede hacerse fácilmente mediante el trazado de una línea que comienza en el origen de la cámara y que pasa por el centro de cada píxel. Se puede expresar esta línea en la forma de un rayo cuyo origen es el origen de la cámara y cuya dirección es el vector desde el origen de la cámara en el centro del píxel. Para calcular la posición de un punto en el

centro de un píxel, se necesita convertir las coordenadas de píxeles que se expresa originalmente en el espacio de la trama al espacio del mundo.

El espacio mundial es básicamente el espacio en el que todos los objetos de la escena, la geometría, las luces y las cámaras tienen sus coordenadas expresadas. Por ejemplo, si un disco se encuentra a cinco unidades de distancia del origen del mundo a lo largo del eje z negativo las coordenadas espaciales mundiales del disco son (0, 0, -5). Si se quiere que las matemáticas calculen la intersección de un rayo con este disco para trabajar, el origen y la dirección del rayo también necesitan ser definidos en el mismo espacio. Por ejemplo, si un rayo tiene origen (0, 0, 0) y la dirección (0, 0, -1) en las que estos números representan las coordenadas en el espacio del sistema de coordenadas, entonces el rayo se cruzará con el disco en (0,0,-5).

Lo primero a considerar y que será de mucha ayuda, es conocer cómo se componen el archivo que carga la escena. Este es un ejemplo de una típica escena:

Lo primero será definir el tamaño que tendrá la imagen

```
image_size 640 480
```

Lo siguiente es definir el campo de visión (field of view)

```
field_of_view 60
```

Las variables de la cámara, las cuales definen:

1) Su posición

2) La dirección donde mira la cámara (donde se encuentra la cuadrícula)

3) Y por último, donde se encuentra arriba según la dirección de la cámara.

```
camera_position 0.0 0.0 0.0

camera_look 0.0 1.0 0.0

camera_up 0.0 0.0 1.0
```

*El tercer dato es muy importante, se coloca el valor de camara_arriba según los otros dos valores, en caso de que camara_arriba sea el valor incorrecto, la creación de la imagen será errónea, en este ejemplo el valor correcto puede ser cualquier valor en el eje “x” o “z”, al igual que una combinación de ambos, según como se requiera que la cámara obtenga la imagen. Esto se debe a que la cámara está en el origen y se encuentra viendo hacia el eje “y”.

Ahora se definirá una figura, seguido del material *posición* en el que está hecho, la posición en donde se encuentra y el radio para esferas o distancia para el plano (para mayor información revisar las prácticas pasadas)

```
sphere 0 -1.0  1.5  0.5  0.5

plane  1  0.0  0.0  1.0  0.0
```

Por último es la palabra material (según la posición que se colocan, es el número de material que se le da las figuras pasadas), seguido del color (red, green, blue), y diversas variables como: difuso, reflexión, brillantes, etc. (color se revisó en prácticas pasadas, las demás variables se revisarán en prácticas posteriores)

```
Material 1.0 0.4 0.0 0.9 0.8 0.2 0.5 0.0 0.0

Material 0.8 0.8 0.9 1.0 0.0 0.4 0.1 0.0 0.0
```

Una vez revisado lo que un archivo de texto debería contener, lo siguiente a considerar es abrir y leer dicho archivo. Pero lo primero es definir unos arreglos

Además de inicializar algunos arreglos que ayudarán a contener información de los objetos y materiales de la escena.

```
objects = Array.new()
materials = Array.new()
image = Image.new(320, 240) (valor predeterminado)
fov = 60 (valor predeterminado)
Open_file(filename)
      line = for each word[i]; i++ (arreglo de palabras)
```

```
	case line[0]
		when "#"
			Next
		when "field_of_view"
			fov = line[1].convert_to_float
		when "image_size"
			image.width = line[1].convert_to_integer
			image.height = line[2].convert_to_integer
		when "camera_position"
			cam_pos = parse_vector(line[1 to 3])
		when "camera_look"
			cam_look = parse_vector(line[1 to 3])
		when "camera_up"
			cam_up = parse_vector(line[1 to 3])
		when "material"
			materials.append(parse_material(line))
		when "plane"
			objects.append(plane.new
			(line[1].convert_to_integer,parse_vector(line[2 to
			4]),line[5].convert_to_float))
		when "sphere"
			objects.append(sphere.new(line[1].convert_to_integer,parse_vector
			(line[2 to 4]),line[5].convert_to_float))
		else
			next
	end
end
```

Continua el cálculo de las variables una vez leído el archivo, lo siguiente es calcular el grid o cuadrícula a través de la cual se trazaron los rayos, un ejemplo de esto se puede

ver en la Figura 2.4.4. (*Esta sección será utilizada para el oversampling, que veremos en prácticas futuras).

```
grid_width = image.width

grid_height = image.height
```

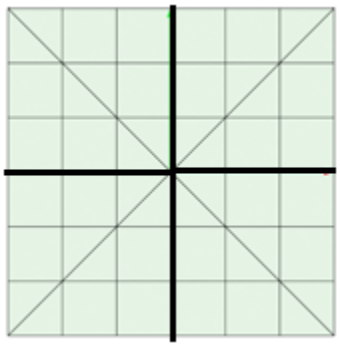


Figura 2.4.4 Cuadrícula por donde se pasarán los rayos

Lo siguiente es el cálculo sobre los vectores de la cámara para trazar la pirámide formada por la cuadrícula y el origen de la cámara, esto se puede ver en la imagen Figura 2.4.5.

```
look = cam_look - cam_pos (recuerda que ambos son vectores)
```

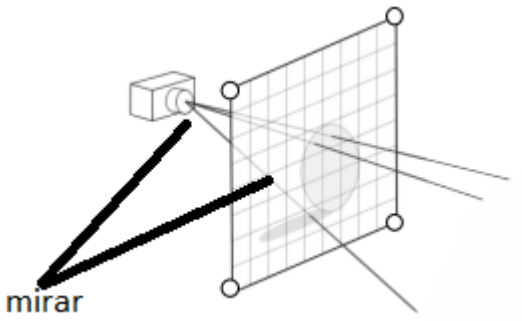


Figura 2.4.5 Cálculo de "mirar"

Ahora se calculara vhor y vver los cuales son vectores que definen la vista horizontal y vertical, después se normalizara, ambos se calculan usando producto cruz para obtener los vectores perpendiculares.

```
vhor = look.cross(cam_up)  (producto cruz)

Vhor.normalize

vver = look.cross(vhor)  (producto cruz)

vver.normalize
```

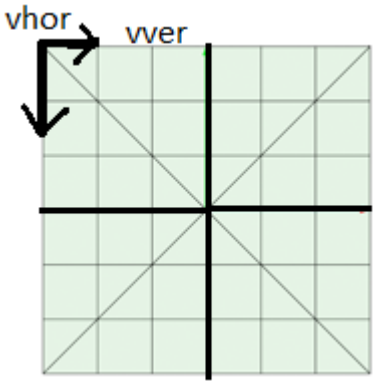


Figura 2.4.6 vhor y vver

Se normaliza después mirar

```
vp = look

vp.normalize
```

Por último se realizan los últimos cálculos en la cuadrícula

```
fl = grid_width / (2 * Math.tan((0.5 * fov) *  3.1415/180))
```

Vale la pena detacar la configuración de la cámara desde la vista lateral, se puede dibujar un triángulo uniendo el origen de la cámara en el borde superior e inferior del plano de la película. Porque se sabe que la distancia desde el origen de la cámara en el plano de la película y la altura del plano de la película se puede utilizar algo de trigonometría simple para encontrar el ángulo del triángulo rectángulo ABC que es la mitad del ángulo vertical del campo de visión, esto se puede ver en la Figura 2.4.7.

Por lo tanto, se multiplican las coordenadas del pixel de pantalla por este número para escalar hacia arriba o hacia abajo. Como se puede suponer, esta operación cambia la forma en gran parte de la escena que se ve, y es equivalente a zoom (se ve menos de la escena en la que el campo de visión disminuye) y fuera (y ver más de la escena en la que el valor del campo de visión aumenta). En conclusión, se puede definir el campo de visión de la cámara en función del ángulo "a", y multiplicar el píxel de la pantalla se coordina con el resultado de la tangente de este ángulo dividido por dos.

```
vp.x = vp.x * fl - 0.5 * (grid_width * vhor.x + grid_height * vver.x)

vp.y = vp.y * fl - 0.5 * (grid_width * vhor.y + grid_height vver.y)

vp.z = vp.z * fl - 0.5 * (grid_width * vhor.z + grid_height * vver.z)
```

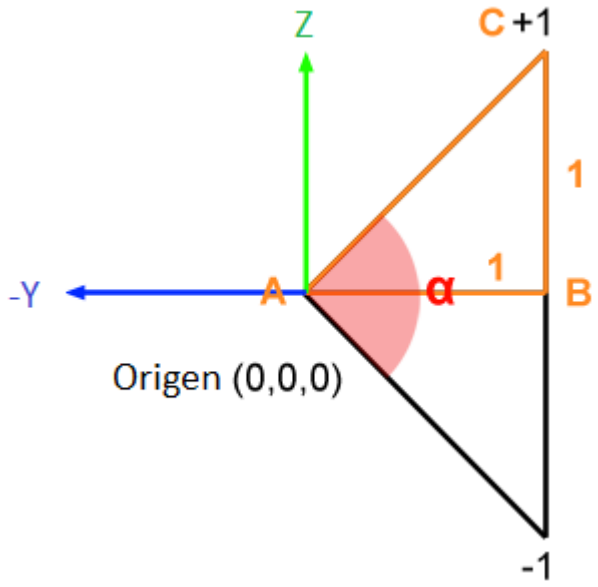


Figura 2.4.7 Cálculo usando el campo de vista

Por último se agregan las funciones de "parseo" del archivo que se utilizará en la parte de arriba. Esto quiere decir que convertirá los valores en flotantes utilizando la función respectiva, por ejemplo, se tomarán los tres valores del vector y se utilizara la función vector definida en la práctica 1.

```
def parse_vector(line)

      return Vector3.new(line[0].convert_to_float, line[1].convert_to_float,
      line[2].convert_to_float)

end

def parse_color(line)

      return   Color.new(line[0].convert_to_float,   line[1].convert_to_float,
      line[2].convert_to_float)

end

def parse_material(line)

      f = line[4 to -1].map{|l| l.convert_to_float}

      return Material.new(parse_color(line[1 to 3]), f[0], f[1], f[2], f[3],
      f[4], f[5])
```

```
	end
```

Pseudocódigo

```
class Scene
	objects = Array.new()
	materials = Array.new()
	image = Image.new(320, 240) (valor predeterminado)
	fov = 60 (valor predeterminado)
	Open_file(filename)
		line = for each word[i]; i++ (arreglo de palabras)
		case line[0]
			when ("#")
				next
			when ("field_of_view")
				fov = line[1].convert_to_float
			when ("image_size")
				image.width = line[1].convert_to_integer
				image.height = line[2].convert_to_integer
			when ("camera_position")
				cam_pos = parse_vector(line[1 to 3])
			when ("camera_look")
				cam_look = parse_vector(line[1 to 3])
			when ("camera_up")
				cam_up = parse_vector(line[1 to 3])
			when ("material")
				materials.append(parse_material(line))
			when ("plane")
				objects.append(plane.new(line[1].convert_to_integer,parse_v
				ector(line[2 to 4]),line[5].convert_to_float))
			when ("sphere")
```

```
                    objects.append(sphere.new(line[1].convert_to_integer,parse_
                    vector(line[2 to 4]),line[5].convert_to_float))

              else

                    next

       end

end

grid_width = image.width

grid_height = image.height

look = cam_look - cam_pos (recuerda que ambos son vectores)

vhor = look.cross(cam_up)  (producto cruz)

Vhor.normalize

vver = look.cross(vhor)  (producto cruz)

vver.normalize

vp = look

vp.normalize

fl = grid_width / (2 * Math.tan((0.5 * fov) *  3.1415/180))

vp.x = vp.x * fl - 0.5 * (grid_width * vhor.x + grid_height * vver.x)

vp.y = vp.y * fl - 0.5 * (grid_width * vhor.y + grid_height vver.y)

vp.z = vp.z * fl - 0.5 * (grid_width * vhor.z + grid_height * vver.z)

def parse_vector(line)

       return    Vector3.new(line[0].convert_to_float,line[1].convert_to_float,
       line[2].convert_to_float)

end

def parse_color(line)

       return   Color.new(line[0].convert_to_float,   line[1].convert_to_float,
       line[2].convert_to_float)

end

def parse_material(line)

       f = line[4 to -1].map{|l| l.convert_to_float}

       return Material.new(parse_color(line[1 to 3]), f[0], f[1], f[2], f[3],
       f[4], f[5])

end
```

```
end
```

Pruebas del código

Para poder revisar el código de esta práctica, se realizará pruebas enfocándonos en la lectura del archivo y su correcto guardado.

Para ello se creará un archivo escena txt en donde se llenará con una escena sencilla de raycaster, que el código reconocerá y guardará adecuadamente para entonces imprimir los arreglos en los que fueron guardados.

Para este ejercicio crear el siguiente documento:

```
image_size 700 500

field_of_view 50

cameraPos  5.0 5.0 5.0

cameraLook 0.0 0.0 0.0

cameraUp   0.0 0.0 1.0

sphere 0 2.5 0.0 0.0 1.0

plane  1 0.0 0.0 1.0 0.0

material 0.5 0.6 1.0 0.1 0.7 0.5 0.3 1.0 0.4

material 0.7 0.8 0.2 0.2 1.0 0.7 0.3 0.2 0.1
```

*Se recuerda que se requiere de las prácticas anteriores para poder realizar esta prueba.

Tabla de pruebas

Caso	Entrada	Salida

Imprimir un conjunto de arreglos donde se guardaron los valores del archivo escena	escena.txt	Imprime los siguientes valores
		`objects`
		`materials`

		`image`
		`fov`
		`cameraPos`
		`cameraLook`
		`CameraUp`
		`grid_width`
		`grid_height`
		Ahora el alumno comparará con los valores del archivo escena, y verificará que estos sean iguales

Tabla 2.4.1 Pruebas

Resultados: *(el alumno colocará aquí cuales han sido los resultados de su práctica).*

Ejemplo: código fuente exclusivamente del seudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto I en la práctica V.

Las impresiones de pantalla de los resultados obtenidos.

Cualquier otra evidencia.

Conclusiones: *(el alumno colocará aquí sus conclusiones por realizar la práctica).*

Ejemplo: Se aprendieron los conceptos fundamentales de que es una escena y como configurarla en un Raytracer. Lo cual es muy importante para poder construir un raytracer…

Bibliografía:

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Kevin Suffern; Ray Tracing from the Ground Up; A K Peters/CRC Press (September 6, 2007)

2.5 Práctica V Trazado de rayos y colisiones con objetos

Objetivo

Los alumnos aprenderán a modelar el trazado de rayos, empleando la técnica de Raytracer y obtendrán la intersección entre algunos vectores que interactúan dentro de la escena, los cuales se interpretarán como colisiones de los objetos con la trayectoria de la luz para poder definir los colores y limitar las áreas o polígonos que se mostrarán dentro de la imagen final que se generará la escena, mediante la aplicación de operaciones con los vectores que representan la luz y el vector normal de los objetos.

Introducción

El método de trazado de rayos es una alternativa eficaz y sencilla que permite calcular de forma unificada la reflexión y la refracción de la luz, sombras, eliminación de superficies ocultas y otros efectos necesarios para conseguir escenas foto realistas, con este método surgieron aproximaciones más completas que resolvían de forma más exacta la ecuación de renderizado, basándose en los mismos principios de trazado de los caminos que sigue la luz.

La idea básica del trazado de rayos es representar la trayectoria de la luz desde sus fuentes emisoras hasta que llegan a la posición del observador. La simulación del camino natural de la luz tiene un inconveniente principal y es que la mayoría de los rayos nunca llegan al observador o plano de la imagen, lo que nos resulta computacionalmente prohibitivo[22].

Para evitar trazar un número alto de rayos que no llegarán al plano de la imagen, y como solución sencilla a la discretización del espacio continuo en píxeles, se emplea el trazado de rayos hacia atrás (backward RayTracing), donde los rayos parten del plano imagen hasta alcanzar los objetos de la escena. La Figura 2.5.1 muestra un esquema general de los componentes básicos que forman un trazador de rayos hacia atrás.

[22] Jhon F. Hughes, Computer Graphics Principles and Practice Tercera Edicion, 2014

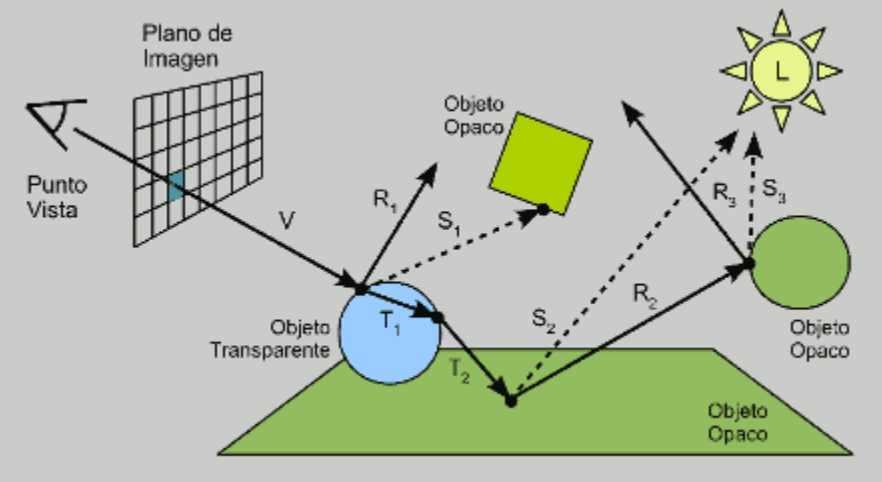


Figura 2.5.1 Esquema de funcionamiento del Raytracing[23]

Una de las principales características del método de Raytracing es el cálculo recursivo de la contribución de la luz debido a la reflexión y refracción que se produce en ciertas superficies. Así, definimos cuatro tipos de rayos:

Rayos Primarios o Visuales. Son los rayos que parten de la cámara virtual, pasando por todos y cada uno de los píxeles en el plano de imagen, además para cada elemento de la escena se comprueba si el rayo intersecta con alguno de ellos, con el que obtenemos el punto de intersección más cercano de toda la lista de objetos.

Rayos de Sombra. Parten del punto de intersección con el objeto y tienen dirección hacia las fuentes de luz, al igual que la categoría anterior se realiza un cálculo de intersección del rayo con todos los objetos de la escena para ver si hay algún objeto que corte su trayectoria cuando el resultado de esa operación es positivo el punto de origen del rayo estaría en sombra.

Rayos Reflejados. Si el objeto donde intersectó el rayo tiene propiedades de reflexión de tipo espejo, se generará un nuevo rayo reflejado en ese punto. Este rayo se construirá típicamente en un procedimiento recursivo, pasando a comportarse como un rayo primario en la siguiente iteración del algoritmo.

[23] https://www.cs.umd.edu/~mount/Indep/MJump/report.html University of Maryland

Rayos Transmitidos. En el caso de objetos en mayor o menor grado transparentes, y de forma análoga al tratamiento para los rayos reflejados, se generará un rayo transmitido, de igual forma, este nuevo rayo se comportará como un rayo primario en la siguiente iteración del algoritmo.

Colisiones. En los procesos de desarrollo de gráficos asistido por computadora, las colisiones juegan un papel fundamental tanto que sin ellas se puede decir que no hay nada que se pueda visualizar en la imagen, principalmente en el área de los videojuegos.

Con esto ya se tiene casi todo hecho, solo es necesario sumar los radios de los círculos y restarles la distancia obtenida, si la distancia es <= a 0 entonces existe colisión.

Sin embargo esta fórmula tiene una debilidad y es que usada de manera muy frecuente durante la ejecución del programa en lapsos muy cortos se vuelve poco eficiente, así que en su lugar se puede hacer uso de la siguiente implementación, que si bien no brinda un resultado 100% preciso, si ofrece una aproximación a este y sobre todo es mucho más rápida que la versión anterior, esta implementación hace uso de la Serie de Maclaurin , con cálculos que resultan computacionalmente más fáciles de hacer, lo que permite aprovechar al máximo el procesamiento de los dispositivos para la generación de la imagen final.[24]

Desarrollo:

Para iniciar se recordará que en otras prácticas anteriores ya se han utilizado algunos elementos, los cuales aunque se argumenta en esta práctica, son necesarios para la correcta implementación de otras prácticas, recordar como fue implementada la clase Rayo que será una de las clases más importantes dentro del desarrollo.

1. Se declara la clase rayo, la cual al ser un vector, tienen un origen y una dirección.

Además se almacena la distancia de alguna intersección cuando el vector rayo colisione con un objeto de la escena, así como la referencia del objeto con el que tenga la colisión.

[24] Jhon F. Hughes, Computer Graphics Principles and Practice Tercera Edicion, 2014

2. En el constructor de la clase, sólo se define el origen y la dirección del rayo, sin embargo, la distancia inicial será una constante definida a nivel global del programa de la escena, y el objeto de colisión es nulo cuando se crea el rayo, es por esto que la constante se define previamente a nivel global, la cual se interpretará en la vida real como un horizonte de ubicación.

Es importante que la constante MAX_DISTANCE sea igualada a un número muy alto como 99999999999, para que al iniciar las comparaciones en la ejecución del programa, cualquier valor real sea menor y pueda conservar el valor de esa primer comparación.

3. También se creará una función que pueda definir si es que existe intersección con los planos de la trayectoria del rayo que se esta modelando, para el caso de las esferas es necesario calcular un vector que vaya desde el centro de la esfera hasta el origen del rayo trazado.

Mediante el cálculo del producto punto se calcula la distancia que hay entre el rayo y la esfera si se resta el radio de la esfera y esa distancia es mayor a la del rayo, la esfera no es el objeto más cercano así que se puede ignorar.

En caso contrario, se resuelve la ecuación cuadrática para revisar la colisión entre el rayo y el plano de corte de la esfera.

En caso de que sea negativa se esta viendo una cara interior de la esfera, la cual tendría dos interpretaciones, la primera es que se este viendo la parte trasera de la misma o bien la cámara se encuentra dentro de la esfera.

return false if inter < 0.0

Como ya se sabe que hay intersección entonces se calcula la distancia del rayo al plano en que el rayo colisiona con la esfera, si la distancia de intersección es negativa, la colisión se encuentra detrás de la cámara, así que se ignora, del mismo modo si la distancia de colisión es mayor a una que el rayo haya encontrado antes se descarta pues no es el objeto más cercano.

Si nada de lo anterior se cumple, la colisión es la más cercana hasta ahora, así que se guarda en el rayo la distancia y la referencia del objeto con el que tiene la colisión.

4. Se declara la función de intersección del plano, la cual revisa si hay colisiones entre el rayo trazado y el plano, en caso de existir dicha intersección se asignan valores al rayo, para ello se calcula el producto punto entre la normal del plano y la dirección del rayo, tomando en cuenta que un rayo siempre colisionará con un plano a menos de que sean paralelos, si el producto punto es 0 significa que el ángulo entre el rayo y la normal del plano tiene un valor de 90°. Lo cual implica que el rayo y el plano son paralelos y nunca colisionarán entre sí.

Se calcula la distancia de intersección entre el origen del rayo y la superficie del plano,- si la distancia de intersección es negativa, la colisión se encuentra detrás de la cámara, así que se ignora, del mismo modo si la distancia de colisión es mayor a una que el rayo haya encontrado antes se descarta pues no es el objeto más cercano.

En caso de que ningún criterio anterior se cumpla, la colisión es la más cercana hasta ese momento, así que se guarda en el rayo la distancia y la referencia del objeto con el que hubo colisión, ya que es una colisión válida.

Ahora, habiendo retomado estos 4 puntos importantes, se puede continuar con el desarrollo de la práctica de trazado de rayos y colisiones con objetos, para la cual se implementa la clase Raytracer, esta contendrá todo lo necesario para generar la imagen final a partir de la descripción de escena.

Los métodos importantes son trace(ray) que se dedica a trazar rayos y render_pixel(x, y) que es la que renderiza el pixel en esas coordenada de la imagen. La función render es el ciclo principal de renderizado.

1. El constructor es igual al de la escena pues sólo se encarga de construirla y empezar el proceso, guardando el nombre para utilizarlo como nombre de la imagen que se va a generar.

2. La función trace será la función que emita los rayos y revise las colisiones contra los objetos de la escena, posteriormente se convertirá en una función recursiva para el

trazado de rayos y recibirá el parámetro 'depth' o la profundidad de trazado para poder detener el proceso.

Se itera sobre todos los objetos de la escena y se prueba la intersección del rayo con cada uno, es importante considerar que en caso de colisión el rayo tendrá la referencia al objeto más cercano y la distancia de colisión con el objeto.

3. Se revisa si hubo intersección para aplicar el color del objeto, el objeto guarda el índice de la lista de materiales que se le hayan aplicado, y busca el color que le corresponde para poder aplicarlo.

Para los casos en que no haya intersección se aplicará el color deafult (negro)

4. La función render_pixel(x, y) crea los rayos a partir de la cámara, al mandar llamar trace y almacena el color final en la imagen en las coordenadas x, y, se calcula la dirección del rayo interpolando en la cuadricula de la cámara.

Como el origen del rayo siempre será la posición de la cámara, se guarda el color que se obtuvo en la imagen.

Usando Render como ciclo principal, donde se itera sobre toda la imagen y después se guarda el resultado, se renderiza cada pixel.

5. Y para concluir se guarda la imagen que se generó y se verifica que la imagen se haya creado de manera correcta.

Pseudocódigo

```
Ray
  attr_accessor :origin, :direction
  attr_accessor :origin, :direction, :distance, :object
  def initialize(origin, direction)
    origin = origin
    direction = direction
    distance = MAX_DISTANCE
```

```
    object = nil
  end
end
  def intersect(ray)
    sph_ray = center - ray.origin
    v = sph_ray.dot(ray.direction)
    return false if v - radius > ray.distance
    inter = radius**2 + v**2 - sph_ray.x**2 - sph_ray.y**2 - sph_ray.z**2
    inter = v - Math.sqrt(inter)
    return false if inter < 0.0
    return false if inter > ray.distance
    ray.distance = inter
    ray.object = self
    return true
  end
end

  def intersect(ray)
    v = normal.dot(ray.direction)
    return false if v == 0.0
    inter = -(normal.dot(ray.origin) + distance) / v
    return false if inter < 0.0
    return false if inter > ray.distance
    ray.distance = inter
    ray.object = self
    return true
  end
end

class Raytracer
```

```
def initialize(filename)
  filename = filename
  scene = Scene.new(filename)
end
def trace(ray)
  scene.objects.each do |obj|
    obj.intersect(ray)
  end
def render_pixel(x, y)
  direction = Vector3.new()
  direction.x = x * scene.vhor.x + y * scene.vver.x + scene.vp.x
  direction.y = x * scene.vhor.y + y * scene.vver.y + scene.vp.y
  direction.z = x * scene.vhor.z + y * scene.vver.z + scene.vp.z
  direction.normalize
  ray = Ray.new(scene.cam_pos, direction)
  scene.image.data[y*scene.image.width+x] = trace(ray)
end

def render()
  (0...scene.image.height).each do |y|
    print "Rendering line %.3d of %d\r" % [y+1, scene.image.height]
    (0...scene.image.width).each do |x|
      render_pixel(x, y)
    end
  end
```

Pruebas.

La prueba sería:

El programa será capaz de generar trazado de rayos y mostrar su comportamiento al momento de realizar colisiones con los objetos que se encuentran dentro de la escena, con lo cual se verifica que los objetos se iluminen con cada elemento de la escena.

Tabla de pruebas

Caso	**Entrada**	**Salida**
Colisión con círculos	Esferas: material posición y radio Esfera 1 3 3.0 1.5 0.8 1.0	

Tabla 2.5.1 Pruebas

Resultados: *(el alumno colocará aquí cuales han sido los resultados de su práctica).*

Ejemplo: código fuente exclusivamente del seudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto I en la práctica V.

Las impresiones de pantalla de los resultados obtenidos.

Cualquier otra evidencia.

Conclusiones: *(el alumno colocará aquí sus conclusiones por realizar la práctica).*

Ejemplo: Se aprendieron los conceptos fundamentales de que es el color y que es una imagen dentro de la computación gráfica. Lo cual es muy importante para poder construir un raytracer…

Bibliografía:

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

2.6 Proyecto I Implementación de un Raytracer básico

Objetivo

Implementar un raytracer básico que emita rayos, revise colisiones con los objetos de una escena, obtenga el color del objeto más cercano a la cámara y guarde una imagen con el resultado.

Introducción

El proyecto implica la suma de todas las prácticas anteriores para generar un programa funcional que reciba el nombre de un archivo de texto con una descripción de escena y produzca una imagen con el mismo nombre con el resultado de haber implementado el algoritmo correctamente. Es necesario notar que, al ser una implementación básica, todavía no se aplican shaders ni la naturaleza recursiva del algoritmo. Esta implementación realiza entonces ray casting (emisión de rayos) para obtener únicamente el color de los objetos de la escena, sin importar sus demás propiedades, por lo cual en el resultado final no podrá observarse una representación fiel en 3D de la escena. Sin embargo, es esencial realizar esta implementación básica del algoritmo para asegurarnos que el funcionamiento de nuestro código es el correcto. Las pruebas previas en cada práctica nos aseguran que cada segmento de código se comporta como debe, y en este proyecto pondremos en funcionamiento todas las partes en una sola aplicación monolítica. A continuación, se presenta la Tabla 2.6.1 con todas las clases, variables y métodos con los que se cuentan hasta el momento para asegurarnos de contar con los elementos necesarios para la realización del proyecto. *Nota: Para fines prácticos, no se considera el constructor de cada clase en la tabla.*

Clase	Variables de clase	Métodos de clase
Vector3	x, y, x	dot, cross, module, normalize, +, -, *
Color	r, g, b	+, *, to_string
Image	data, width, height	to_PPM
Material	color, diffuse, specular, shininess, reflect, trasmit, ior	
Ray	origin, direction, distance, object	
Object3D	type, material	
Sphere	center, radius	get_normal, intersect
Plane	normal, distance	get_normal, intersect
Scene	objects, materials, image, fov, cam_pos, cam_look, cam_up, vhor, vver, vp, look	parse_vector, parse_color, parse_material
Raytracer	scene, filename	trace, render_pixel, render

Tabla 2.6.1 Listado de clases, variables y métodos

Una vez asegurando que el código cumple con todos los elementos para proceder con la implementación del proyecto, es necesario comprender como se suman todos los elementos para poder generar la imagen final a partir de nuestro algoritmo. Partiendo del diagrama funcional del algoritmo mostrado en el capítulo 1, podemos simplificar nuestro procedimiento para este proyecto con el siguiente diagrama.

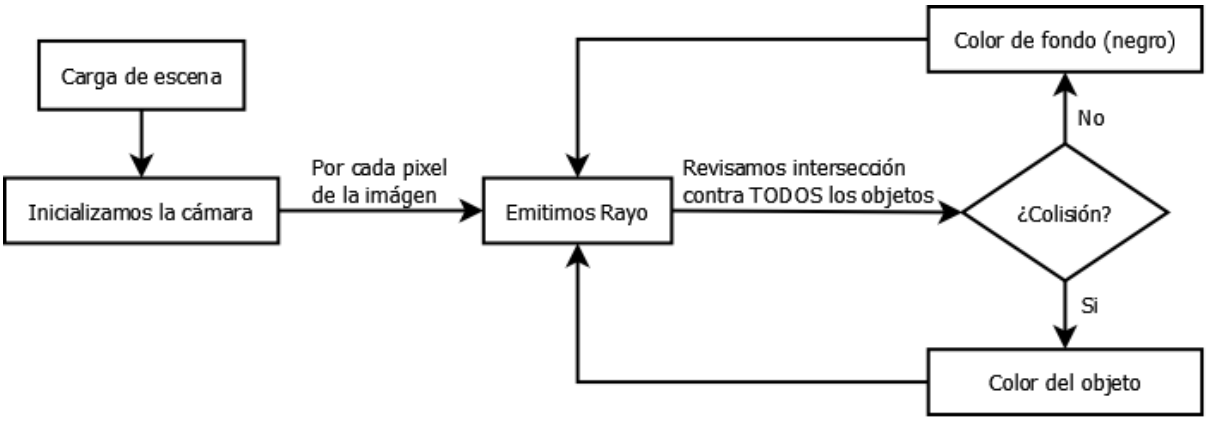


Figura 2.6.1 Diagrama de flujo de la implementación

En la Figura 2.6.1 se puede observar como se procederá con el algoritmo comenzando con la carga de escena desde un archivo de texto con la descripción, después se calculan los vectores que componen a la cámara y la matriz de visión de la imagen, y por último se ejecuta el ciclo principal iterando sobre cada pixel de la imagen emitiendo rayos y buscando colisiones. En el caso de esta implementación se puede observar cómo se simplifica considerablemente la obtención del color final y podemos asegurar que sólo se emitirá un solo rayo por pixel de la imagen y un solo chequeo de colisión por objeto de la escena por cada rayo emitido. El resultado será una representación plana de los objetos, sin sombreado y sin ningún tipo de técnica de iluminación, lo cual permite asegurar que los métodos de colisión y emisión de rayos funcionan correctamente.

Desarrollo

La clase *Raytracer* es la que incluye el ciclo principal en su método *render*, para lo cual se tiene que construir un objeto de esta clase en la función *main* del programa. El constructor del objeto ***Raytracer*** recibe el nombre del archivo con la descripción de escena y con él crea un objeto de tipo ***Scene*** cuyo constructor recibe el mismo nombre de archivo, lo abre, lee su contenido y carga el contenido de la escena en el objeto, el nombre del archivo se guarda para posteriormente salvar la imagen con el mismo nombre.

Una vez que se tenga el objeto ***Raytracer*** bastará con llamar su método ***render*** para ejecutar el ciclo principal y esperar a que termine la ejecución del programa para obtener la imagen final como resultado.

El método ***render*** sólo se encarga de iterar sobre las dimensiones de la imagen, y mandar llamar ***render_pixel*** en cada una de las coordenadas (x, y) de la imagen, también imprime en pantalla el avance del proceso pues puede tardar mucho, y por último guarda el resultado final en una imagen en formato PPM. El pseudocódigo del método ***render*** es el siguiente.

```
iteramos y desde 0 hasta alto de imagen - 1
    imprime "Avance: línea %d de %d", y + 1, alto de imagen
    iteramos x desde 0 hasta ancho de imagen - 1
```

```
                render_pixel(x,y)
    guardamos la imagen final en formato PPM
```

Para probar la implementación se parte de una escena muy sencilla con fines de prueba. Se ubica la cámara sobre el eje coordenado **Y** viendo hacia el origen. Se coloca un plano sobre el plano **XZ** y cuatro esferas con centro sobre el plano, una en cada cuadrante. Posteriormente se agregan tres materiales base a la escena, los cuales serán los colores primarios rojo, azul y verde. Es importante mencionar que, aunque todavía no se utilizan los parámetros adicionales de cada material, ya está implementada su lectura y carga, así que es importante definirlos en la descripción de escena, para fines prácticos todos estos valores serán cero. Por último, se utilizan unas dimensiones de imagen cuadradas y el valor de campo de visión predeterminado que es de 60 grados. Se asigna el color azul al plano, y los colores rojo y verde a las esferas en cuadrantes opuestos.

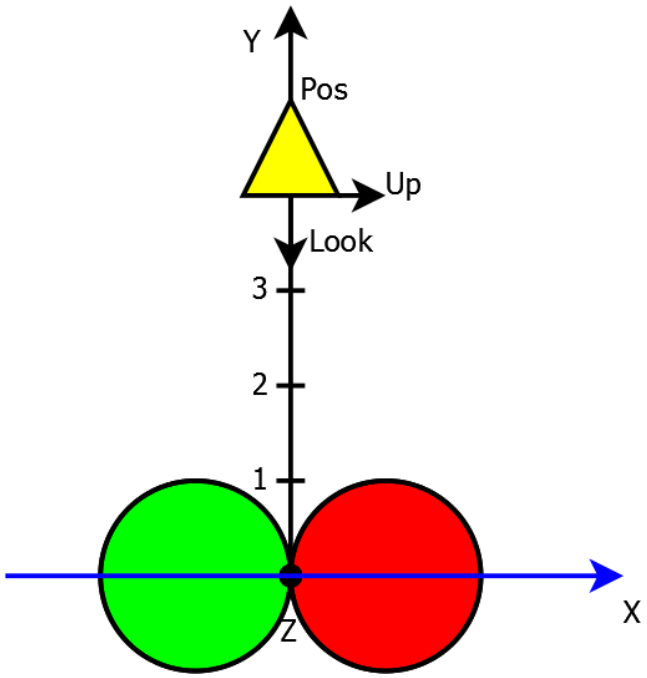


Figura 2.6.2

Se puede observar un diagrama de la escena en la figura 2.6.2, en el cual se ve la proyección del plano **XY** donde sólo son visibles dos esferas, que ocultan aquellas en los cuadrantes del eje **Z** negativo con colores opuestos. Para poder obtener la imagen final,

el archivo de descripción de escena queda de la siguiente manera. *Nota: Todas las líneas que comienzan con # son consideradas comentarios y son ignoradas.*

```
# Dimensiones de la imagen(x,y) se utiliza alto = ancho para una imagen cuadrada.
# Cabe mencionar que mientras más grande sea, más tiempo tardará el proceso.
image_size 512 512
# Campo de Visión, 60° de apertura de cámara es el predeterminado (opcional)
field_of_view 60
# Variables de la cámara todas son: vector(x y z)
# Posición de la cámara sobre el eje Y en 5.0 (POS)
camera_position 0.0 5.0 0.0
# La cámara estará viendo hacia el origen (LOOK)
camera_look 0.0 0.0 0.0
# El vector de orientación, define que "arriba" será el eje X (UP)
camera_up 1.0 0.0 0.0
# material: color(r g b) diffuse specular shininess reflect transmit IOR
# Sólo se modifican los colores RGB en 1.0 para cada color, se ignora el resto 0.0
material 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
material 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
material 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
# sphere: material center(x y z) radius
# Todas las esferas tendrán radio 1.0 colocaremos una en cada cuadrante del
# plano XZ y se asignan los materiales rojo y verde a esferas opuestas.
sphere 0 -1.0 0.0  1.0 1.0
sphere 1  1.0 0.0  1.0 1.0
sphere 1 -1.0 0.0 -1.0 1.0
sphere 0  1.0 0.0 -1.0 1.0
# plane: material normal(x y z) distancia
# El plano coincidirá con el plano coordenado XZ y se le asigna el color azul
plane 2 0.0 1.0 0.0 0.0
```

Se guarda el archivo de escena en formato texto (*.txt) y se inicializa el objeto ***Raytracer*** con el nombre del archivo. Después de ejecutar el método ***render*** se debe de obtener una imagen como la siguiente.

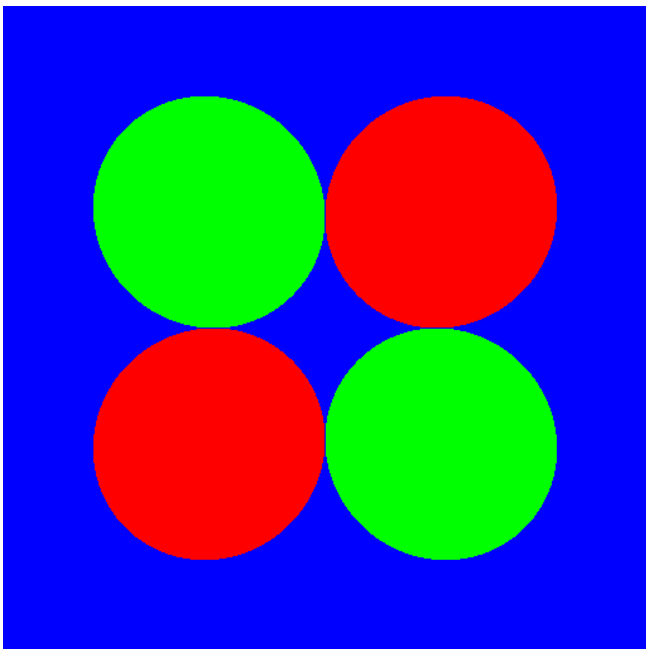


Figura 2.6.3 Primera imagen del Raytracer

Se puede apreciar en la figura 2.6.3 el resultado final del trazado de rayos y como se esperaba, se ve una representación plana y sin sombreado de la escena 3D, es preferible empezar con una proyección de este tipo pues facilita encontrar los errores con una escena muy sencilla. Para asegurar que el trazado de rayos funciona correctamente y se perciba que en efecto se está tratando con una descripción de escena en tres dimensiones, se modificarán algunos parámetros de la escena para poder apreciar mejor las esferas en corte con el plano.

Se mueve la cámara hacia la posición [5.0, 5.0, 5.0] y es muy importante entender que el vector **UP** de la figura 2.6.2 se tiene que recalcular correctamente o se obtendrán resultados inesperados. El vector **UP** se encuentra en el origen y apunta en la dirección que la cámara considera arriba, siempre tiene que ser perpendicular a **LOOK** pues son los encargados de formar el sistema coordenado de nuestra cámara, modificando el vector **UP** se puede rotar la cámara sobre su propio eje de visión **LOOK** sin embargo, si

está mal calculado y no son perpendiculares se obtendrán incongruencias y distorsiones en la imagen. Como se sigue viendo hacia el origen, el vector **POS** es la identidad en **XYZ** positivas con magnitud 5.0, el vector **UP** correspondiente se puede deducir de la inversa de este, **XZ** serán negativas, mientras que **Y,** se conserva positiva, es importante notar que **UP** se normaliza en los cálculos y su magnitud es irrelevante, por lo cual obtendremos [-1.0, 1.0, -1.0] como vector final. Por último, para ajustar el *"zoom"* de la escena, se cambia el valor del ángulo del campo de visión (*field of view*) a 45 para ver la escena más de cerca. Al final se modificaron las siguientes líneas del archivo de descripción de escena. *Nota: Se pueden guardar estas modificaciones como un nuevo archivo de texto para tener una escena nueva.*

```
# Campo de Visión, 45° de apertura para ver más cerca
field_of_view 45
# Posición de la cámara sobre la identidad en: (POS)
camera_position 5.0 5.0 5.0
# El vector de rotación, define que "arriba" será el eje X (UP)
camera_up -1.0 1.0 -1.0
```

Con estos cambios se ejecuta el programa principal y se obtiene como resultado la imagen mostrada en la figura 2.6.4 en la cual se pueden apreciar con mayor detalle las esferas, así como el corte del plano sobre **XZ** y la rotación que se le hizo a la cámara. Esta escena, aunque simple, permite corroborar el comportamiento del Raytracer básico el cual es capaz de crear una imagen a partir de la descripción virtual de la escena trazando rayos a partir de la cámara.

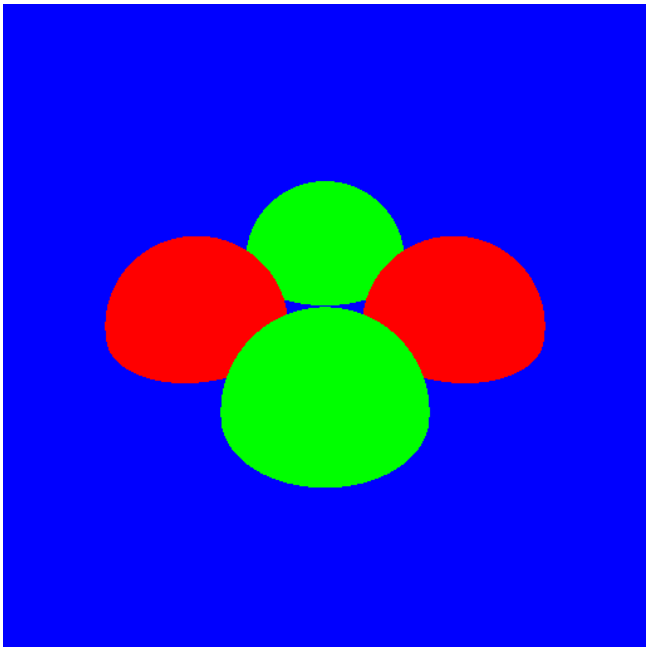


Figura 2.6.4 Raytracing de una escena simple

Es muy importante el funcionamiento que se tiene, pues hasta este punto se suele tener la mayor carga de código escrito, para el capítulo 3 las modificaciones al programa serán casi exclusivamente en las funciones de **trace** y **render_pixel**, donde se incluyen el cálculo de luces, sobras, shaders, reflexión y refracción. Ya no se regresará a modificar ninguna funcionalidad básica más que la carga de luces desde el archivo de escena o algunos parámetros adicionales. Es muy recomendable que se experimente en este punto moviendo la cámara, modificando la escena, o bien creando un programa que anime la cámara o los objetos en la escena. Cuando se esté familiarizado con el funcionamiento del programa y del algoritmo del Raytracing, se puede proceder al capítulo 3 donde se verán conceptos más avanzados de computación gráfica.

Se puede encontrar una implementación de referencia en su última versión en:

https://git.io/vDUFC

3 CAPÍTULO III. CONCEPTOS AVANZADOS DE ILUMINACIÓN Y RENDERING

3.1 Práctica VI Conceptos fundamentales de iluminación y sombras

Objetivo

El alumno será capaz de entender los conceptos de iluminación, como se maneja la luz en el algoritmo de raytracer. También entenderá el concepto de sombras. Partirá del conocimiento físico de que es la luz y la sombra para de ahí entender cómo funciona en el algoritmo. Entendiendo estos conceptos y siguiendo el seudocódigo será capaz de codificar el programa respectivo de la práctica.

Introducción

La luz es un tópico demasiado común en nuestras vidas porque siempre estamos en contacto con ella. Por lo tanto es un tema que ha sido tratado de entender desde el principio de la humanidad. Teorías de diversa índole fueron y vinieron. Alguna predicó que nuestros ojos eran los que crean la luz en el mundo, que en el momento que abríamos los ojos lanzábamos rayos de luz sobre el mundo que veíamos. En la actualidad sabemos que eso es un error pero en ciertos casos puede ser útil ese tipo de pensamiento. Hoy en día tenemos varias teorías sólidas de cómo funciona la luz, y nos permiten manipularla y tener múltiples aplicaciones.

Óptica. Rama de la física que se encarga del estudio de la luz así como de los fenómenos luminosos. La interacción con los elementos físicos del mundo. Aquí se encuentran diferentes acercamientos al fenómeno. La luz puede ser estudiada como una partícula o como una onda electromagnética. También se puede realizar estudios desde la física cuántica. La luz como una partícula o una onda electromagnética son enfoques útiles para nuestros fines. Todas las teorías actuales son válidas aunque se contradigan entre sí en algunos aspectos. Funcionan en su delimitación del conocimiento.

Luz (partícula). La óptica puede analizar a la luz como un rayo. La propagación de la luz puede ser estudiada en términos de "rayos" que viajan en línea recta, y cuyos caminos se rigen por las leyes de la reflexión y refracción en las interacciones con diferentes objetos materiales.

Cuando un rayo de luz incide en un material transparente, este se divide, reflejando y refractando el rayo. La ley de la reflexión dice que un rayo reflejado está en el plano de

incidencia y el ángulo de reflexión es igual al ángulo de incidencia. La ley de la refracción dice que cuando un rayo refractado choca en el plano de incidencia, se puede obtener una constante *n,* ésta resulta de la división del seno del ángulo de refracción entre el seno del ángulo de incidencia. Se le llama índice de refracción del material. Figura 3.1.1

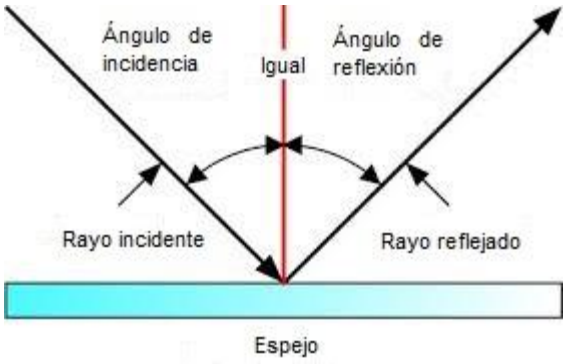


Figura 3.1.1 La luz como partícula

Luz (onda electromagnética). En este caso, la luz se considera como una onda electromagnética la cual se propaga en un medio. Este modelo predice fenómenos tales como la interferencia y la difracción, que no se explican mediante el modelo en que la luz es una partícula. La velocidad de las ondas de luz en el aire es de aproximadamente 3,0 x 108 m / s (exactamente 299.792.458 m / s en el vacío). La longitud de onda de las ondas luminosas visibles varía entre 400 y 700 nm. Cualquier luz con una longitud de onda inferior a 380 nm o por arriba de 740 nm no pueden ser percibidas por el ojo humano.

El modelo de onda se puede utilizar para hacer predicciones acerca de cómo un sistema óptico se comportará sin requerir una explicación de cómo el medio está siendo perturbado. La existencia de las ondas electromagnéticas se predijo en 1865 por las ecuaciones de Maxwell. Estas ondas se propagan a la velocidad de la luz y tienen diferentes campos eléctricos y magnéticos que son ortogonales entre sí, y a la dirección de propagación de las ondas. Una onda periódica se define por su frecuencia (el número de veces en un ciclo que se repite por unidad de tiempo). La longitud de onda es el inverso de la frecuencia. El color de la luz puede ser visto como el equivalente del concepto de tono para el sonido. Ambos se basan en la longitud de onda o frecuencia de

la señal que viaja a través del espacio. El modelo de la luz como una onda electromagnética es muy usado hoy en día a excepción cuando otro modelo es más pertinente. La longitud de onda se denota con la letra griega λ (lambda). Figura 3.1.2

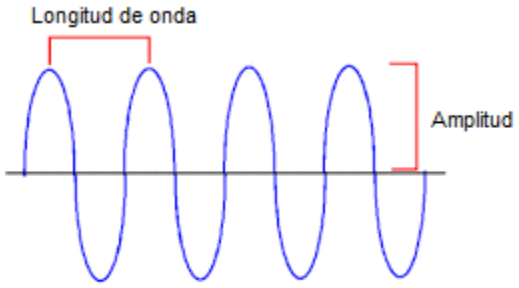


Figura 3.1.2 La luz como onda

Luz (fotones). Descrita como una cadena de partículas llamadas fotones. Este nombre se lo dio Albert Einstein. Fundamentalmente cuando se estudia la luz desde este acercamiento es para hacer un análisis de la luz interactuando con la materia en microscópicos niveles.

Luz blanca. Esta hecha de todos los colores visibles del espectro de luz visible, mezcladas en diferentes proporciones. La luz blanca, como tal, no existe. Es el resultado de una fuente de luz, el sol u otra fuente artificial, produciendo una mezcla de colores claros a partir del espectro visible. Cuando se puede observar detenidamente y de forma amplificada se ve que la luz blanca está compuesta por la mezcla de los colores básicos.

Composición de la luz blanca. La luz blanca está compuesta por los colores básicos cada uno tiene su propio espectro:

- Rojo (618-780 nm)
- Anaranjado (581-618 nm)
- Amarillo (570-581 nm)
- Verde (497-570 nm)

- Cian (476-497 nm)
- Azul (427-476 nm)
- Violeta (380-427 nm)

El color de los materiales. En el mundo sólo es posible el fenómeno de los colores debido a que existe la luz, sin ésta sería imposible hablar de este tópico. El color surge de la interacción de la luz con algún material. Como ya mencionamos la luz puede ser considerada como fotones de los siete colores básicos. Al chocar la luz con un material este absorbe alguno de los 7 posibles rayos de color. El rayo que no es absorbido, es el color que tendrá el material en cuestión. Por ejemplo, en el caso que un rayo de luz incida sobre un material y todos los colores sean absorbidos menos el amarillo, este es rebotado. Nuestro ojo humano percibirá que el material es de color amarillo. Si el material rebota varios colores de luz, dependiendo de cuales sean las luces rebotadas formaran un color en particular. Si el material rebota todos los colores el ojo observará el color blanco. En el caso donde el material absorba todos los rayos de luz. Se Percibirá el material como de color negro.

Colores primarios. Dependiendo del campo, se puede establecer unos colores primarios u otros. De cualquier modo es la combinación de los colores primarios la que da por resultado los demás colores. En la computación, como quedó establecido en la práctica II, los colores primarios son el rojo verde y azul. En cuanto a la luz, las luces son el color amarillo, azul y rojo. Las dos definiciones de colores básicos sirven para un propósito específico de utilización en el campo de interés. Retomando el concepto de luz blanca, que podría producirse a partir de la suma de luces de colores. Se tiene una luz de un determinado color y al añadir otra luz de otro color, se producirá un tercer color del espectro visible.

Iluminación del algoritmo Raytracer. Se observa de las definiciones anteriores que el fenómeno de la luz es un tema muy complicado el cual implica que programarlo, de la forma en que la física considera a luz, sería sumamente complejo. Por lo que resulta poco viable seguir completamente los modelos de definición de la luz de la física. El algoritmo

Raytracer busca simplificar toda la cuestión matemática para presentar un modelo en donde se pueda manipular la luz de una forma más fácil y eficiente. Por supuesto que la convención que se adopta para el manejo de la luz dentro de la creación de imágenes digitales por medio del algoritmo Raytracer sólo es válida en esta delimitación (computación gráfica). Sería un error grave usar este tipo de acercamiento a la luz en cualquier otro contexto.

En la figura 3.1.3 se observa como es el fenómeno de luz en el mundo real basados en las teorías aceptadas. Sería demasiado complejo para la computación gráfica trabajarlo de esa manera.

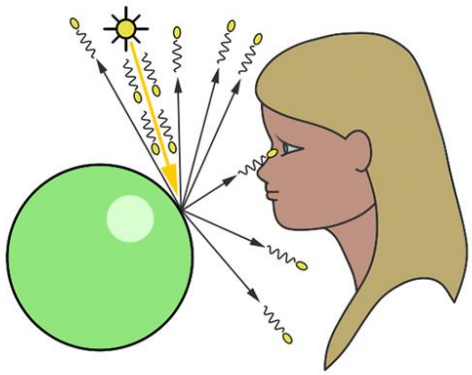


Figura 3.1.3 La luz como rayos de fotones[25]

Por lo cual es preferible seguir un enfoque mucho más simple y que hace que resulte lo más sencillo modelarlo. Esto es el caso de la simplificación que hace el algoritmo Figura 3.1.4.

[25] http://www.scratchapixel.com/

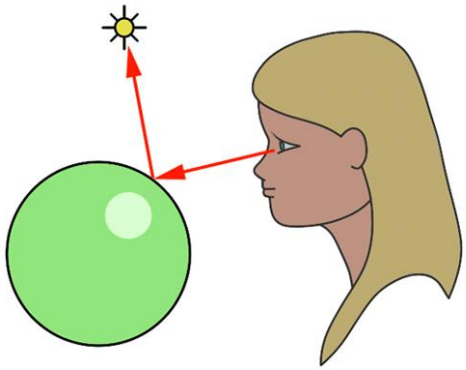


Figura 3.1.4 La luz como rayos bajo el modelo de Raytracer[26]

De la figura se observa como el rayo sale desde el punto de visión, este sigue una trayectoria recta hasta que choca con un objeto, en este caso una esfera, en ese momento el rayo se desvía en forma perpendicular a la fuente de luz. Miles de rayos son disparados por el punto de visión (cámara) para crear la iluminación del objeto.

Sombras del algoritmo Raytracer En el caso que el rayo pegue en la parte inferior del objeto, se observa que el rayo no puede rebotar para crear una perpendicular hacia la fuente la luz. Por lo que el algoritmo determina que a esa superficie no le llega la luz directamente por lo que el algoritmo en esa parte de la superficie existirá la sombra. También existe el caso en donde el rayo rebota perpendicular en la superficie dirigiéndose hacia la fuente de luz, pero algo se interpone y el rayo no llega a la fuente de luz. Figura 3.1.5. Esto implica que hay un objeto que se interpone y que este generará una sombra directamente debajo sobre la superficie más cercana. Como podemos ver el modelo de sombras también es un poco distinto a cómo funciona en la realidad. En el mundo existe una sombra porque muy pocos o ningún rayo de luz llega a determinada superficie. En el caso del sombreado del raytracer, las sombras existen, debido a que los rayos que salen del punto de visión (cámara) no pueden llegar a la fuente de luz, por lo cual el algoritmo infiere que existe una sombra en la superficie. Esto se puede observar en la figura 3.1.5 las sombras bajo el modelo del Raytracer.

http://www.scratchapixel.com/

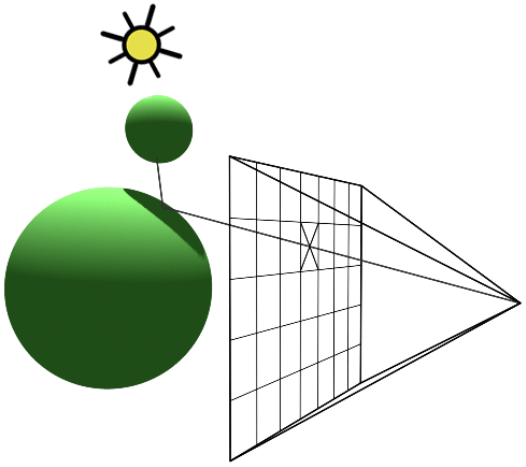


Figura 3.1.5 las sobras bajo el modelo de Raytracer[27]

Recapitulando, el algoritmo ilumina (luz) o oscurece (sombra) una superficie dependiendo del rayo que viaja desde la cámara hasta el objeto y de ahí rebota perpendicularmente hacia la fuente de luz. Si el rayo llega a la fuente de luz, el algoritmo ilumina la superficie donde chocó el rayo. Si el rayo no llega a la fuente de luz debido a un objeto. El algoritmo ensombrece la superficie donde chocó el rayo. Pueden existir algunos casos en donde el objeto que intersecta el rayo desde el objeto a la fuente de la luz, sea de material con un índice de transparencia. En ese caso, como sucedería en la física el rayo continuaría, pero también el algoritmo crearía una sombra de acuerdo al tipo de material del objeto interceptor. Como se puede ver el algoritmo trabaja el asunto de las luces y sombras de una forma sencilla y elegante que es mucho más fácil de programar que si se toma el modelo exacto que define la física.

Desarrollo

Después de entender toda la teoría referente a la iluminación y las sombras, el alumno codificará el seudocódigo. Es preferible que utilice un lenguaje orientado a objetos, aunque también se puede utilizar un lenguaje de programación estructurada.

Las ventajas del lenguaje orientado a objetos es que se definirá un objeto (vector, luz, color, material etc.) cada objeto con sus respectivas funciones. Como en ocasiones se va

[27] http://www.scratchapixel.com/

a operar a nivel de objetos, de ahí la conveniencia de utilizar un lenguaje orientado a objetos. Si se utiliza un lenguaje estructurado tendrá que manejarlo todo por funciones.

Pseudocódigo

```
Iluminación
Inicio
	Definimos la clase luz con tres parámetros type, position, color
	Inicializamos la clase
	Asignamos las variables para que puedan ser usadas por el rayo
	Parámetro type = type
	Parámetro position = position
	Parámetro color = color
Fin
En la clase escena se tiene que mandar a llamar el método light
lights = ArrayNew

Sombras

Inicio

	Definimos la clase para calcular las sombras
	Recibe los datos a partir del rayo generado, dato: punto de colisión, objeto que
	colisionó
	Regresa la atenuación de color [0.0. 1.0] Or la sombra
	El valor de Shadow = 1.0 debido a que no hay sombra
	Si choca con otro objeto
	Multiplicamos la sombra por la opacidad del objeto colisionado. Se utiliza el
	parámetro transmit para el cálculo de la sombra.

Fin

Calculo de los parámetros de la luz y las sombras

Inicio
	Extraemos el material del objeto colisionado
```

```
        Cálculos =  material del objeto colisionado
        Calculamos el punto de intersección con el objeto
        Inter_point = rayOrigin + rayDirection * rayDistance
        Vector incidente
        Incident = inter_point - rayOrigin
        Vector en sentido opuesto al rayo
        backOrigin = rayDirection * -1.0
        Calculo de la normal en el punto de colisión
        InterNormal = Normal en el punto de intersección
        Iteramos sobre las fuentes de luz
        Si la luz es de tipo ambiental, sumamos el color de la luz al color resultado
        Si la luz es de tipo puntual...
        Calculamos el vector que va de la luz hacia el punto de intersección con el
        objeto para ver si hay sombra
        Se lanza un nuevo rayo que del punto de intersección hacia la fuente de luz para
        revisar colisiones con otros objetos
        Se llama al objeto cálculo de la sombra
        Se regresa un  valor de sombra

Fin
```

Pruebas del código.

La prueba sería:

- El programa será capaz de iluminar o sombrar las superficies con las que choque el rayo. Ver imagen 3.1.6 Imagen de prueba iluminación y sombras

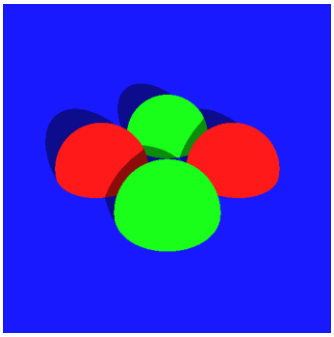


Figura 3.1.6 Imagen de prueba iluminación y sombras

Tabla de pruebas

Caso	Entrada	Salida
Iluminación y sombras	Archivo de la escena	Archivo de imagen con iluminación y sombras. Ver figura 3.1.6

Tabla 3.1.1 Pruebas

El alumno podrá poner la fuente de luz en otras coordenadas y observar como las sombras varían. También podrá poner más fuentes de luz.
Se comienza a codificar basándonos en el seudocódigo.

Resultados: *(el alumno colocará aquí cuales han sido los resultados de su práctica). Ejemplo: código fuente exclusivamente del seudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto II.*
Las impresiones de pantalla de los resultados obtenidos.
Cualquier otra evidencia.

Conclusiones: *(el alumno colocará aquí sus conclusiones por realizar la práctica). Ejemplo: Se aprendieron los conceptos fundamentales de que es la luz. Además como se maneja el fenómeno de la luz en el algoritmo que es distinto a como funciona en el mundo real.*

Bibliografía:

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

3.2 Práctica VII Sombreado de superficies

Objetivo

El alumno comprenderá el concepto del sombreado de una superficie o Shading, que será la forma en la cual el raytracer colocara la sombra a nuestros objetos de la escena. Para ello se partirá con la iluminación con Phong y llegando hasta su algoritmo. Al término de la teoría se verá el seudocódigo con su respectiva explicación, para que al final de a practica el alumno logre codificar y anexar esta sección a su código completo del raytracer.

Introducción

La iluminación en un punto es igual a la suma de la contribución de cada fuente de luz individual. El problema aquí es, en una manera simplificada, proporcionando una lista finita de luces del punto bien definidas.

Eliminamos rápidamente cualquier luz que no sea visible desde nuestro lado de la superficie. Eso es si la salida normal a nuestra figura y la dirección de la luz, apuntan hacia la dirección opuesta.

Entonces determinamos si el punto está en la sombra de otro objeto para esta luz particular.

Se lanzará un rayo desde nuestro punto de intersección hacia la fuente de luz (es posible porque es un punto de luz, que es un solo punto en el espacio). Pero esta vez no nos interesa encontrar lo que el rayo está intersectando o a qué distancia. Tan pronto como encontramos un punto de intersección entre este nuevo rayo y cualquier objeto en la escena, detenemos la iteración porque sabemos que el objeto está en la sombra de luz. Vea Figura 3.2.1

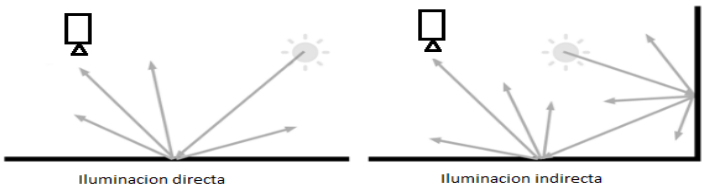


Figura 3.2.1 Tipos de iluminación

Phong

Bui Tuong Phong era un estudiante en la universidad de Utah cuando inventó el proceso que tomó su nombre (iluminación Phong). Es una forma rápida de hacer objetos que reflejan la luz en una dirección privilegiada sin una reflexión completa. En la mayoría de los casos esa dirección privilegiada es la del vector de luz, reflejada por el vector normal de la superficie.

La reflexión de Phong es un modelo empírico de iluminación local. Describe la forma en que una superficie refleja la luz como una combinación de la reflexión difusa de las superficies rugosas con la reflexión especular de las superficies brillantes.

Se basa en la observación informal de Bui Tuong Phong de que las superficies brillantes tienen pequeños reflejos especulares intensos, mientras que las superficies opacas tienen grandes resaltes que se caen de forma más gradual. El modelo también incluye un término ambiental para dar cuenta de la pequeña cantidad de luz que se dispersa en toda la escena, esto se puede ver en la figura 3.2.2.

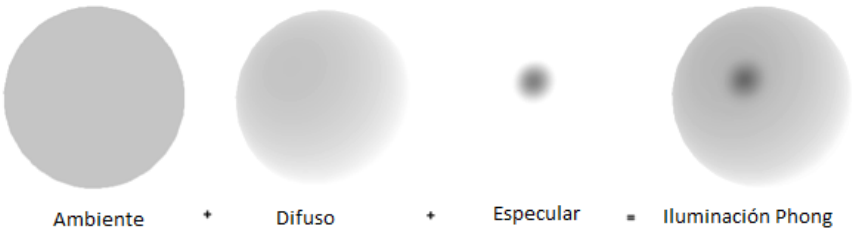


Figura 3.2.2 Iluminación Phong

En este modelo, pensamos que la interacción entre la luz y una superficie tiene tres componentes distintos:

-Ambiente
-Difuso
-Especular

Normalmente el componente ambiente recibe un valor constante tenue, tal como 0,2. Se aproxima a la luz procedente de una superficie debido a toda la luz ambiental no

direccional que se encuentra en el entorno. En general, se requiere que este sea un poco de color teñido, en lugar de sólo gris. [r_a, g_a, b_a]. Por ejemplo, un objeto ligeramente verdoso podría tener un color ambiente de [0.1, 0.3, 0.1].

El componente difuso es ese producto punto **n • L** . Se aproxima a la luz, originalmente de la fuente de luz **L**, que se refleja desde una superficie que es difusa o no brillante. Un ejemplo de una superficie no brillante es el papel. En general, también querrá que tenga un valor de color no gris, por lo que este término sería en general un color definido como: [r_d, g_d, b_d] (**n • L**).

Por último, el modelo Phong tiene una disposición para un componente destacado, o especular, que refleja la luz de una manera brillante. Esto es definido por [r_s, g_s, b_s] (**R • L**) p, donde **R** es el vector de dirección de reflexión de espejo que hemos discutido en clase (y también usado para el trazado de rayos), y donde p es una potencia especular. Cuanto más alto sea el valor de **p**, más brillante será la superficie.

El modelo completo de sombreado Phong para una sola fuente de luz es:

$$[r_a, g_a, b_a] + [r_d, g_d, b_d] \max0 (\mathbf{n} \bullet \mathbf{L}) + [r_s, g_s, b_s] \max0 (\mathbf{R} \bullet \mathbf{L})^p$$

Si tiene varias fuentes de luz, el efecto de cada fuente de luz Li dependerá geométricamente de la normal, y por lo tanto de los componentes difusos y especulares, pero no del componente ambiental. Además, cada luz puede tener su propio color [r, g, b]. Así que el modelo completo de Phong para múltiples fuentes de luz es:

$$[r_a, g_a, b_a] + \Sigma_i ([L_r, L_g, L_b] ([r_d, g_d, b_d] \max0 (\mathbf{n} \bullet \mathbf{L_i}) + [r_s, g_s, b_s] \max0 (\mathbf{R} \bullet \mathbf{L})^p))$$

Difuso y especular

El sombreado difuso es parte de Phong, un material difuso dispersa la luz en todas las direcciones, por lo que su fuente más brillante está exactamente allí donde el material se

enfrenta a la fuente de luz. Tomando el producto punto entre la normal y un vector a la luz da este resultado.

El sombreado especular es un poco diferente. Básicamente, el resalte especular es una reflexión difusa de la fuente de luz. Se puede comprobar esto en la vida real al tomar un objeto brillante, ponerlo sobre una mesa bajo una lámpara, y mover tu vista. Al hacer esto se observará que el punto brillante no permanece en la misma posición cuando se mueve, puesto que es básicamente una reflexión, su posición cambia cuando cambia el punto de vista. Phong sugirió el siguiente modelo de iluminación, que realmente toma en cuenta el vector reflejado, esta es la versión más simplificada de dicho modelo:

Intensidad = difuso * (L • N) + especular * (V • R) n

(Donde L es el vector desde el punto de intersección a la fuente de luz, N es la normal en el punto, V es la dirección de la vista y R es L reflejado en la superficie)

Desarrollo

Tomamos la variable de light_dir obtenida en la práctica pasada (esta variable se obtuvo a partir de nuestra definición de dirección de luz, que se calcula con la resta de uno de los atributos de la luz, posición en la que la luz se coloca, menos el punto de intersección con el objeto, que después se vectorizo) y calculamos el producto punto con la normal en el punto de colisión (InterNormal) también definido en la práctica pasada.

```
nl = inter_normal.dot(light_dir)
```

Si la variable nl es mayor a 0.0, quiere decir que la normal en el punto de colisión y la dirección de luz no son vectores perpendiculares y podemos calcular las componentes difusa y especular. En caso contrario significa que la luz es tangente al punto, por lo cual la luz no añade sombreado al punto.

```
if nl > 0.0
```

Todo material se le coloca una componente de difusión que se definió en el archivo de escena en la práctica 4. Se verifica que sea mayor a cero.

```
if mat.diffuse > 0.0
```

Creamos una nueva variable llamada diffuse_color, el cual se calcula multiplicando el color de la luz (light.color), la componente de difusión del material (mat.diffuse) y el recién calculado producto punto entre la normal y la luz en el punto de intersección (nl).

```
diffuse_color = light.color * mat.diffuse * nl
```

Lo siguiente es agregar a este último color difusión, el color del material y la sombra, individualmente a cada componente de color RGB, al multiplicar dichas componentes de las pasadas prácticas.

```
diffuse_color.r *= mat.color.r * shadow
diffuse_color.g *= mat.color.g * shadow
diffuse_color.b *= mat.color.b * shadow
```

La variable de color que se tiene (c) le sumamos también la variable de color difusión (diffuse_color)

```
c += diffuse_color
```

Similar al punto 3, todo material se le coloca una componente especular que se definió en el archivo de escena en la práctica 4. Se verifica que sea mayor a cero.

```
if mat.specular > 0.0
```

El modelo Phong tiene una disposición para un componente destacado o especular, por lo que se calcula esta componente, empezando con el producto cruz del vector en sentido opuesto de la superficie (back_origin) y “r” es la dirección de la vista

```
r = (inter_normal * 2 * nl) - light_dir
spec = back_origin.dot(r)
```

Si el producto punto (especular) es 0.0, significa que los vectores son perpendiculares lo que implica que el punto de colisión es tangente a la iluminación, por ese motivo ignoramos los siguientes.

```
if spec > 0.0
```

El valor especular se eleva a mat.brillo, donde (mat = material) el cual es una componente del material que se definió en prácticas pasadas, esta es una potencia especular. Cuanto más alto sea el valor de mat.brillo, más brillante será la superficie. Después se multiplica por la componente especular del material (mat.specular) y se guarda el resultado en la misma variable especular.

spec = mat.specular * spec**mat.shininess

Y creamos nuestro color especular agregando la sombra y el color de la luz a la variable especular

specular_color = light.color * spec * shadow

A la variable de color (c) le sumamos también la variable de color especular (specular_color)

c += specular_color

Pseudocódigo

```
nl = inter_normal.dot(light_dir)
if nl > 0.0
	if mat.diffuse > 0.0
		diffuse_color = light.color * mat.diffuse * nl
		diffuse_color.r *= mat.color.r * shadow
		diffuse_color.g *= mat.color.g * shadow
		diffuse_color.b *= mat.color.b * shadow
		c += diffuse_color
	end
	if mat.specular > 0.0
		r = (inter_normal * 2 * nl) - light_dir
		spec = back_origin.dot(r)
			if spec > 0.0
			spec = mat.specular * spec**mat.shininess
			specular_color = light.color * spec * shadow
			c += specular_color
		end
	end
```

```
end
```

Pruebas

Para esta sección de prueba se crearan imágenes con objetos con valores en difusión, especular y brillantez que se definen en los materiales. Se requiere de la práctica anterior, ya que se agregaran luces ambientales y puntuales.

Para este ejercicio por favor crea la siguiente escena:

```
size 1024 768

vision 60

cameraPos 6.0 0.0 6.0
cameraLook 0.0 0.0 0.0
cameraUp 0.0 0.0 1.0

sphere 0 -4.0 0.0 0.0 1.
sphere 1 0.0 -3.0 0.0 1.00
sphere 2 2.0 0.0 0.0 1.0
sphere 3 0.0 1.0 0.0 1.0

plane 1  0.0 0.0 1.0  0.0

light -2.0 1.0 4.0  1.0 1.0 1.0 point
light 2.0 13.0 4.0  0.5 0.5 0.5 point
light 0.0 0.0 4.0   0.1 0.1 0.1 ambient

material 1.0 0.4 0.0   1.0 1.0 1.0 0.0 0.0 0.0
material 1.0 0.2 0.2   0.5 0.5 0.5 0.0 0.0 0.0
material 0.2 1.0 0.2   0.2 0.2 0.2 0.0 0.0 0.0
material 0.3 0.3 1.0   0.2 0.5 0.0 0.0 0.0 0.0
material 0.9 0.9 0.9   1.0 0.0 0.0 0.0 0.0 0.0
```

*Recuerdo que se requiere de las prácticas anteriores para poder realizar esta prueba.

Tabla de pruebas

Caso	Entrada	Salida
Creación de una escena con Shading	escena.txt	Ver Figura 3.2.3

Tabla 3.2.1 Pruebas

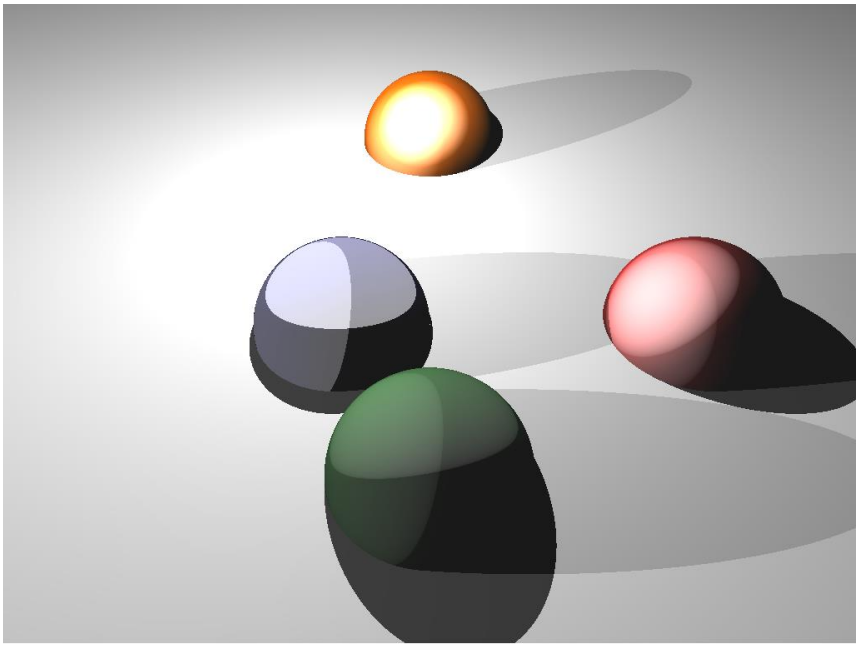


Figura 3.2.3 Pruebas

Resultados

(el alumno colocará aquí cuales han sido los resultados de su práctica).

Ejemplo: código fuente exclusivamente del seudocódigo proporcionado en la práctica. El programa será una parte fundamental para poder construir el proyecto II en la práctica VII.

Las impresiones de pantalla de los resultados obtenidos.

Cualquier otra evidencia.

Conclusiones

(el alumno colocará aquí sus conclusiones por realizar la práctica).

Ejemplo: Se aprendieron los conceptos fundamentales de que es el color y que es una imagen dentro de la computación gráfica. Lo cual es muy importante para poder construir un raytracer…

Bibliografía

Eric Haines, Pat Hanrahan, Robert L. Cook, James Arvo, David Kirk, Paul S. Heckbert; An Introduction to Ray Tracing (The Morgan Kaufmann Series in Computer Graphics) (1st Edition); Academic Press; 1st edition (February 11, 1989)

Peter Shirley; Ray Tracing in One Weekend (Ray Tracing Minibooks Book 1); Amazon Digital Services LLC

3.3 Práctica VII Reflexión y refracción

Objetivo

El alumno comprenderá los conceptos de reflexión y refracción, su aplicación en la computación gráfica e implementará un programa en el lenguaje de su interés que describa los efectos a los que se someten los rayos de luz al impactar sobre una superficie.

Introducción

Se iniciará explicando un concepto que está asociado a los fenómenos de reflexión y refracción, éste es la luz.

Luz. Es una forma de radiación electromagnética, llamada energía radiante, capaz de excitar la retina del ojo humano y producir, en consecuencia, una sensación visual. La energía radiante fluye en forma de ondas en cualquier medio con una dirección determinada (propagación rectilínea), y sólo es perceptible cuando interactúa con la materia, que permite su absorción o su reflejo. Hay entonces un cuerpo emisor de la energía radiante y otro que la recibe. Esta interacción o transferencia de energía de un cuerpo a otro se denomina radiación.
La velocidad con que se propaga la luz depende del medio que atraviesa; no es igual en el aire que en el agua. La luz recorre alrededor de 300 000 kilómetros en un segundo.

Físicamente se puede interpretar la luz de 2 maneras, asociadas entre sí:

• Como una onda electromagnética.
• Como un corpúsculo o partícula.

Espectro Electromagnético. Se denomina espectro electromagnético al ordenamiento de la energía radiante según la longitud de onda o la frecuencia. Se extiende desde longitudes de onda de 10^{-16} hasta 105 metros. En el extremo de las frecuencias más altas (onda corta) de mayor energía están los rayos cósmicos (emitidos durante reacciones nucleares). En el otro extremo se encuentran las ondas largas, utilizadas para comunicaciones de radio, que van de unos milímetros a kilómetros de longitud de onda.

Entre estos extremos están los rayos X, los UV (ultravioleta), los visibles y los IRC (infrarrojos). Los últimos tres son los de mayor importancia en el campo de la iluminación. Para medir el espectro visible la unidad de medida más usada es el nanómetro (nm), que equivale a 10^{-9} metros.

Espectro Visible. Es la porción del espectro electromagnético percibida por el ojo humano, y comprende las emisiones radiantes de longitud de onda desde los 380 nm hasta los 780 nm aproximadamente. La luz blanca percibida es una mezcla de todas las longitudes de onda visibles. El espectro visible se puede descomponer en sus diferentes longitudes de onda mediante un prisma de cuarzo, que refracta las distintas longitudes de onda selectivamente.

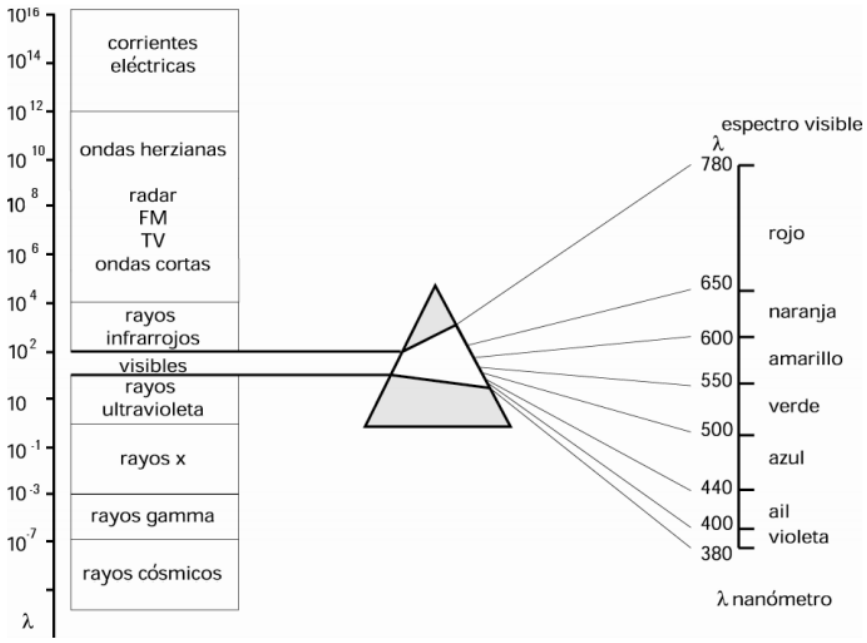


Figura 3.3.1 Espectro electromagnético[28]

[28] http://www.elisirlin.com.ar/11_fisica%20de%20la%20luz.pdf

Rayos primarios.

Considere la siguiente imagen:

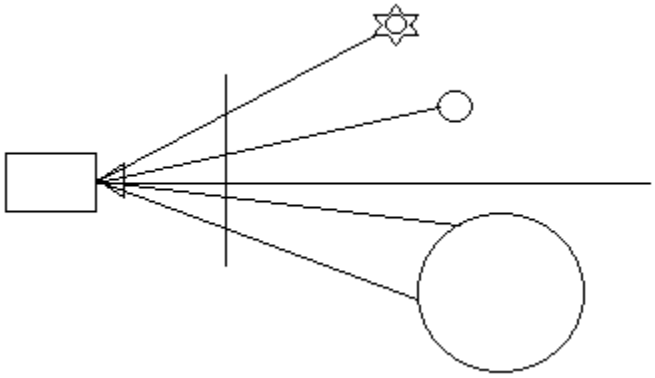


Figura 3.3.2 Emisión de rayos desde cámara virtual

En ésta se muestran los rayos del raytracer simple de la primera sesión de práctica en la escena. Un rayo puede golpear una fuente de luz, o una primitiva geométrica o nada. No hay rebotes ni refracciones. Estos rayos se llaman "rayos primarios".

Además de los rayos primarios, existen los "rayos secundarios". Éstos se muestran en la siguiente figura:

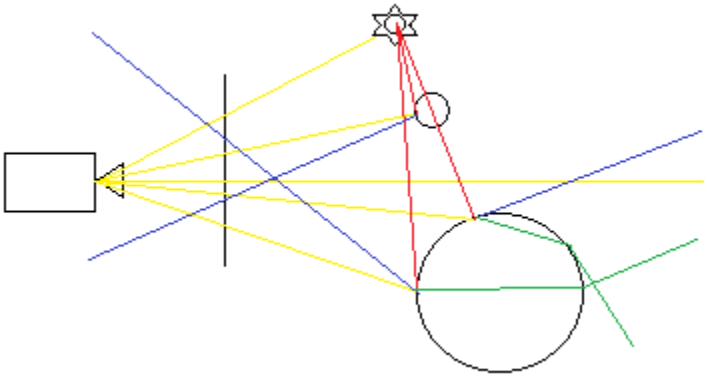


Figura 3.3.3 Varios tipos de rayos secundarios

Las líneas azules en esta imagen son rayos reflejados. Para la reflexión, simplemente rebotan fuera de la superficie. Las líneas verdes son rayos refractados. Estos son un poco

más difíciles de calcular que los rayos reflejados, pero es bastante factible. Implica índices de refracción y una ley formulada por Snell. Las líneas rojas son rayos usados para sondear una fuente de luz. Si se sigue uno de los rayos amarillos que comienzan en la cámara, se notará que cada rayo genera un conjunto de rayos secundarios: Un rayo reflejado, un rayo refractado y un rayo de sombra por fuente de luz. Después de ser lanzado, cada uno de estos rayos (excepto los rayos de la sombra) se trata como un rayo normal. Esto significa que un rayo reflejado puede ser reflejado y refractado repetidas veces. Esta técnica se llama "raytracing recursivo". Cada nuevo rayo se añade al color que reúne su antepasado y, por lo tanto, cada rayo contribuye al color del píxel al que el rayo primario fue originalmente atravesado.

Para evitar bucles sin fin y tiempo de renderización excesivo, normalmente hay un límite en la profundidad de la recursión.

Actualmente existen tres leyes fundamentales para el estudio del comportamiento de la luz, considerando distintas trayectorias posibles de la luz para ir de un punto a otro: propagación directa, reflexión y refracción. En este caso, sólo nos ocuparemos de las dos últimas.

Reflexión. Ocurre cuando las ondas electromagnéticas se topan con una superficie que no absorbe la energía radiante. La onda, llamada rayo incidente se refleja produciendo un haz de luz, denominado rayo reflejado. Si una superficie límite es lisa y totalmente no absorbente se dice que ocurre reflexión especular. En la reflexión especular un solo rayo incidente produce un único rayo reflejado. En el punto de incidencia del rayo incidente, el rayo reflejado y la perpendicular a la superficie límite se encuentran en el mismo plano. El rayo incidente y el rayo reflejado posee ángulos iguales en relación con la perpendicular y se encuentran sobre lados opuestos de ella.

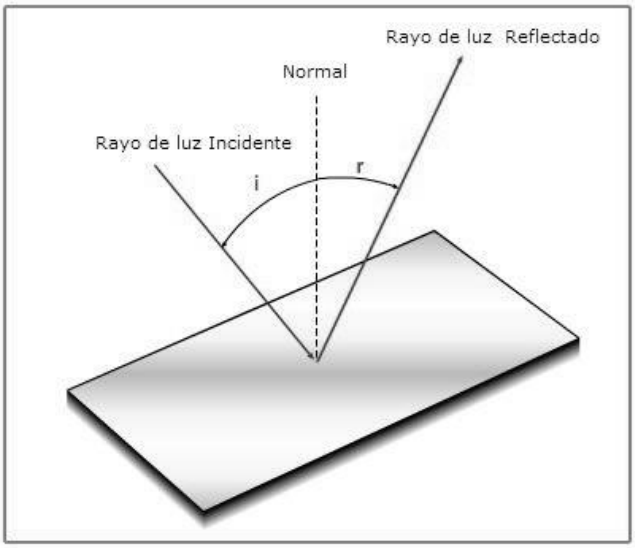


Figura 3.3.4 Reflexión de la luz[29]

La ley de la reflexión establece que: $\hat{i} = \hat{r}$

Donde:

$\hat{i}$ = ángulo de incidencia

$\hat{r}$ = ángulo de reflexión

Refracción. El fenómeno sucede cuando la luz pasa de un medio transparente a otro de diferente densidad y se produce un cambio de dirección debido a la distinta velocidad de propagación que tiene la luz en los diferentes medios materiales.

[29] http://www.portaleducativo.net/tercero-basico/780/La-luz-reflexion-y-refraccion

Figura 3.3.5 Fenómeno de refracción de la luz[30]

La relación entre la velocidad de la luz en el vacío (c) y la velocidad de propagación de una longitud de onda determinada en una sustancia (v) se conoce como índice de refracción (n) de la sustancia para dicha longitud de onda.

$$n = \frac{c}{v}$$

El índice de refracción del aire es 1.00029 y apenas varía con la longitud de onda. Por eso, a la velocidad de la luz en el aire se le considera directamente igual al vacío (valor = 1). En el caso de que la luz pase de aire hacia agua, el índice de refracción relativo es 1.33. Esto quiere decir que la luz es 1.33 veces más rápida en el aire que en el agua. Desde el aire al vidrio común, 1.53. Por lo general, cuando la luz llega a la superficie de separación entre los dos medios se producen simultáneamente la reflexión y la refracción.

[30] http://www.portaleducativo.net/tercero-basico/780/La-luz-reflexion-y-refraccion

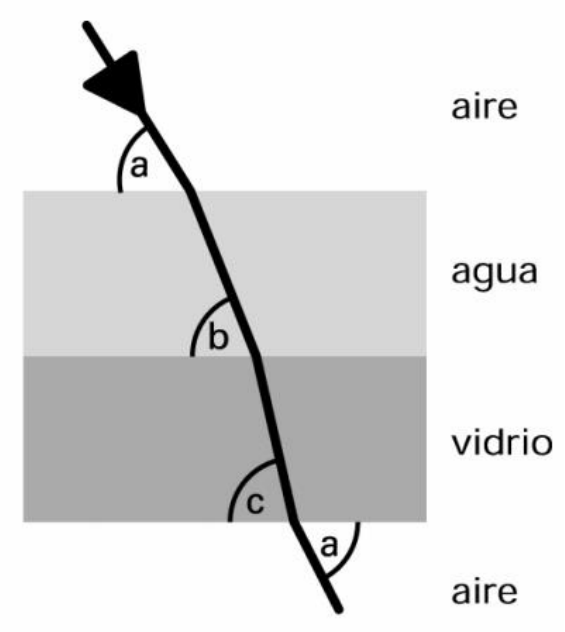


Figura 3.3.6 Refracción de la luz en tres medios diferentes[31]

Índices de refracción. A continuación se presenta una lista con los índices de refracción de algunos materiales.

[31] http://www.elisirlin.com.ar/11_fisica%20de%20la%20luz.pdf

Material	índice de refracción (n)	λ (nm)	Condiciones
Vacío	1		
Aire	1,0002926	589,29	CNPT
Helio	1,000036	589,29	
Oxígeno	1,0002708	667,8	
Hidrógeno	1,000132	589,29	
Dióxido de carbono	1,00045	589,29	
Etanol	1,361	589,29	20,0°C
Agua	1,3330	589,29	20.0°C
Diamante	2,419	589,29	
Ámbar	1,55	589,29	
Ámbar	1,55	589,29	
Hielo	1,31		
Córnea humana	1,3375		
Criolita	1,338		
Acetona	1,36		
Glicerol	1,4729		
Cloruro de sodio (sal)	1,5442	589 nm	
Silicio	4,01		
Zirconia cúbica	2,15 - 2,18		
Teflón	1,35 - 1,38		
Zafiro	1,762–1,778		

Figura 3.3.7 Tabla de índices de refracción[32]

Ley de refracción (Ley de Snell)

Se denomina índice de refracción, al cociente entre la velocidad de la luz c en el vacío y la velocidad v de la luz en un medio material transparente. La relación entre el seno del ángulo de incidencia y el seno del ángulo de refracción es igual a la razón entre la velocidad de la onda en el primer medio y la velocidad de la onda en el segundo medio, o bien puede entenderse como el producto del índice de refracción del primer medio por

[32] https://es.wikipedia.org/wiki/Anexo:%C3%8Dndices_de_refracci%C3%B3n

el seno del ángulo de incidencia es igual al producto del índice de refracción del segundo medio por el seno del ángulo de refracción, esto es:

$$n_1 \cdot sen\theta_1 = n_2 \cdot sen\theta_2$$

- n_1 = índice de refracción del primer medio
- θ_1 = ángulo de incidencia
- n_2 = índice de refracción del segundo medio
- θ_2 = ángulo de refracción

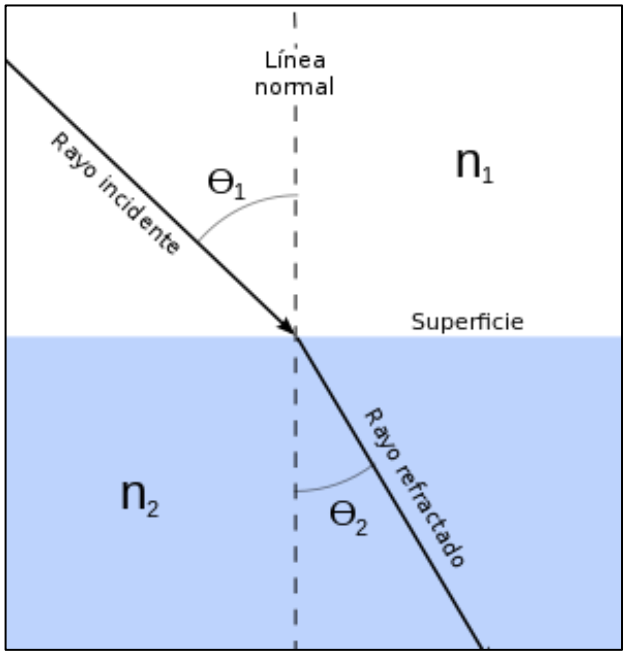


Figura 3.3.8 Refracción de la luz en tres medios diferentes[33]

[33] https://es.wikipedia.org/wiki/%C3%8Dndice_de_refracci%C3%B3n#.C3.8Dndice_de_refracci.C3.B3n_efectivo

Desarrollo

Ahora se aterrizarán los conceptos de reflexión y refracción de rayos de luz sobre primitivas geométricas tal como se aplica en el algoritmo de RayTracer. Observe la siguiente figura:

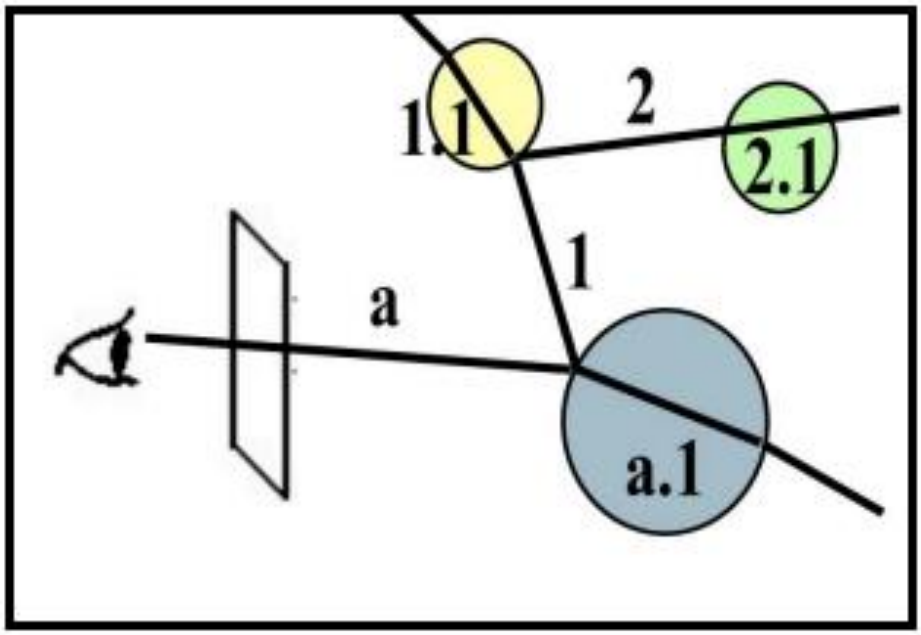


Figura 3.3.9 Rayos reflejados y refractados[34]

En ella se muestra como se lanzarían los rayos reflejados y refractados a partir de una intersección así como la recursión del procedimiento. Como ejemplo se tiene el rayo 1 que es el rayo reflejado a partir del rayo a mientras que a.1 es el rayo refractado a partir del mismo rayo a. El rayo 2 es el rayo reflejado del rayo 1 y el rayo 1.1 es el rayo refractado del mismo rayo y así sigue el proceso sucesivamente para cada primitiva geométrica que se encuentre el rayo.

Es en este momento donde se debe aplicar un algoritmo recursivo, pues dependiendo de las propiedades reflexivas o refractivas del material, se trazan nuevos rayos en el punto de colisión con la superficie y se sigue haciendo esto mientras se encuentren colisiones

[34] http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/carrasco_r_j/capitulo2.pdf

con materiales similares y no se haya alcanzado la profundidad máxima de trazado en nuestro algoritmo.

Todo el procedimiento explicado se repite para cada pixel que forme parte de la imagen final. Por ejemplo, si se quiere analizar una imagen de 800 x 600, entonces habrá que repetir todo el proceso 480,000 veces. De ahí se deriva su lentitud, no obstante, puede producir resultados exactos.

Pseudocódigo

```
Si el objeto refleja (MaterialRefleja>0)
     rayoreflejado = vector anterior.producto punto.vector Normal
     Si rayoreflejado > 0
       vectorDirRef=(vectorNormal * 2 (cte.reflexión) * rayoreflejado) – vector
anterior
       vecInter = puntoInter+vectorDirRef
       rayoRef= vectorRayo (vecInter, vectorDirRef)
       color=vectorTrazar(rayoRef, profundidad + 1) * MaterialRefleja

      color reflejado * cte. de reflexión * rayo reflejado

Si el objeto es transparente (MaterialRefracta>0)
     Rayorefractado=vectorNormal.producto punto.(vectorIncidente*-1)
     Normalizar vectorIncidente
     Si vectorNormal.punto.vectorIncidente > 0
         vectorNormal = vectorNormal * -1
         Rayorefractado = - Rayorefractado
         n1 = IndiceRefraccion
         n2 = 1
     Entonces:
         n2 = IndiceRefraccion
         n1 = 1
     Si n1 y n2 son distintos a 0
         LeySnell = (n1· sen𝜃1 = n2· sen𝜃2)
         vectorDirRefra = vectorIncidente + (vectorNormal*Rayorefractado) * (n1/n2)-
         (vectorNormal * LeySnell)
```

```
vecInter = puntoInter + vectorDirRefra
rayoRefra = vectorRayo(vecInter, vectorDirRefra)
color=vectorTrazar(rayoRefra, profundidad + 1) * MaterialRefracta
```

Pruebas

Caso	Entrada	Salida
Reflexión y Refracción	Archivo de la escena	Archivo imagen donde se vea el fenómeno con la Reflexión y Refracción

Tabla 3.3.1 Pruebas

Resultados

El alumno describirá los resultados obtenidos del desarrollo de su práctica.

Conclusiones

El alumno plasmará sus comentarios que demostrarán que cumplió el objetivo y reafirma el conocimiento adquirido.

Bibliografía:

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

Manual de iluminación, de Eli Sirlin, publicado por el INT, 2005 y Ed. Atuel, 2006;

Fuentes online

http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/carrasco_r_j/capitulo2.pdf

3.4 Práctica IX Oversampling y optimización del código

Objetivo

Que los alumnos aprendan a implementar la técnica del oversampling en la generación de imágenes en tercera dimensión, para que logren diseñar imágenes realistas, debido a la calidad que se puede obtener gracias a esta técnica, que entiendan que pueden utilizar el nivel de oversampling necesario de acuerdo al nivel de precisión que deseen mostrar en sus escenas, además, optimizar el código de sus programas, ya que, dependiendo del nivel de oversampling aplicado el programa se tardará más en resolver, ya que se tienen que hacer 'n' veces más de cálculos para la generación de la imagen, mediante un algoritmo de fácil implementación y comprensible para los interesados.

Introducción

El oversampling (sobre muestreo) es una técnica en la que la frecuencia de muestreo, o en nuestro caso la frecuencia con que se iluminan nuestros pixeles es multiplicada varias veces, el número de muestras o divisiones por unidad de pixel de las que se obtiene el resultado del color que se está iluminando cada pixel de nuestra imagen, para producir una de mejor calidad o de mayor resolución, lo que nos dará como resultado imágenes más realistas debido a que multiplicaremos cuantas veces necesitemos el número de áreas que se iluminaran de colores diferentes, por ejemplo en un sistema con "4x oversampling" la imagen de muestreo es multiplicada en cada pixel por cuatro, lo que nos dará como resultado una imagen de cuatro veces mejor resolución.

Vídeo

En vídeo digital, la frecuencia entre fotogramas es utilizada para definir la frecuencia de muestreo de la imagen en lugar del ritmo de cambios de los píxeles individuales. La frecuencia de muestreo de la imagen es el ritmo de repetición del período de integración del dispositivo de carga acoplado, dado que el periodo de integración puede ser significativamente más corto que el tiempo entre repeticiones, la frecuencia de muestreo puede diferir de la inversa del tiempo de muestreo.

Algunas frecuencias de muestreo típicas en sistemas de audio y vídeo en la tabla 3.4.1

Frecuencias de muestreo típicas	
Para audio	
8000 muestras/s	**Teléfonos, adecuado para la voz humana pero no para la reproducción musical. En la práctica permite reproducir señales con componentes de hasta 3,5 kHz.**
22050 muestras/s	**Radio En la práctica permite reproducir señales con componentes de hasta 10 kHz.**
32000 muestras/s	**Vídeo digital en formato miniDV.**
44100 muestras/s	**CD, En la práctica permite reproducir señales con componentes de hasta 20 kHz. También común en audio en formatos MPEG-1 (VCD, SVCD, MP3).**
47250 muestras/s	**Formato PCM de Nippon Columbia (Denon). En la práctica permite reproducir señales con componentes de hasta 22 kHz.**
48000 muestras/s	**Sonido digital utilizado en la televisión digital, DVD, formato de películas, audio profesional y sistemas DAT.**
50000 muestras/s	**Primeros sistemas de grabación de audio digital de finales de los 70 de las empresas 3M y Soundstream.**
96000 ó 192400 muestras/s	**HD DVD, audio de alta definición para DVD y BD-ROM (Blu-ray Disc).**
2 822 400 muestras/s	**SACD, Direct Stream Digital, desarrollado por Sony y Philips.**
Para vídeo	
50 Hz	**Vídeo PAL.**
60 Hz	**Vídeo NTSC.**

Tabla 3.4.1 Tabla de frecuencias de muestreo[35]

A la hora de renderizar cualquier imagen, podemos realizar ajustes en el campo de "Render", para escoger que tipo de render queremos utilizar y que parámetros para conseguir la mejor relación calidad/tiempo de render.

Dentro del cuadro de diálogo de "Renders" se puede utilizar cualquier parámetro para que se puedan utilizar como punto de partida, y a partir de estos, ajustes preestablecidos iniciales, podemos realizar variaciones en los ajustes que nos ayuden a mejorar la calidad y/o reducir el tiempo de renderizado.

[35] https://es.wikipedia.org/wiki/Frecuencia_de_muestreo

En la imagen 3.4.2 se puede apreciar el resultado de aplicar 9x oversampling a una imagen que da como resultado un realismo esperado, incluso con características mencionadas en prácticas anteriores, como la transparencia el tipo de material entre otras.

Figura 3.4.2 Imagen con 9x oversampling aplicado[36]

Desarrollo

1 Para iniciar con la práctica definiremos los elementos o direcciones del plano sobre el que se creara la escena con oversampling, recordando que podemos tomar cualquier imagen que tengamos elaborada y que se desee aplicar la técnica oversampling para obtener una imagen con mayor realismo, por lo tanto, aplicamos las coordenadas del plano para la generación del oversampling

2 Se necesita calcular la dirección del rayo interpolado en la cuadricula que pertenece a la cámara.

3 En esta sección o parte del programa, es necesario regresar el color que trazamos en nuestra imagen para nuestros próximos cálculos

[36] https://wiki.blender.org/index.php/Doc:ES/2.4/Manual/Materials/Properties/Raytraced_Transparency Blender

4 Generamos el ciclo principal del algoritmo de generación de oversampling donde iteramos sobre toda la imagen, al cual llamaremos ciclo de renderizado y después guardamos el resultado que se obtiene de estas operaciones.

5 Es necesario renderizar cada pixel de la imagen para obtener el resultado que estamos buscando, escribimos en el plano x y nuestros resultados y generamos la imagen final.

Pseudocódigo

```
    c = Color.

    xo = x * scene.oversampling

    yo = y * scene.oversampling

    mientras    scene.oversampling.times

       mientras  scene.oversampling.times

        direction = Vector new

        direction x = xo * scene vh.x + yo * scene.vver.x + scene.vp.x

        direction y = xo * scene.vhor.y + yo * scene.vver.y + scene.vp.y

        direction z = xo * scene.vhor.z + yo * scene.vver.z + scene.vp.z

        Normalizar direction

     end

  def render()

  mientras   (0 hasta scene.image.height)

      escribe  "Rendering line, scene.image.height

      (0 hasta scene.image.width)

        scene.image.data = render_pixel(x, y)

      end
```

Pruebas.

La prueba sería:

El programa será capaz de generar una misma escena con diferentes niveles de oversampling, donde se podrá observar la diferencia entre aplicar y no el método aprendido en este capítulo del curso

Tabla de pruebas

Caso	**Entrada**	**Salida**
Aplicar oversampling a un diseño para observar los cambios en el producto final	Diseño con oversampling 1 (sin oversampling) Aplicando oversampling 3	Referencia Mayor calidad en la imagen

Tabla 3.4.1 pruebas

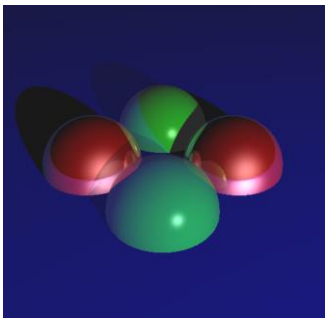


Figura 3.4.3 Imagen con oversampling en valor 1

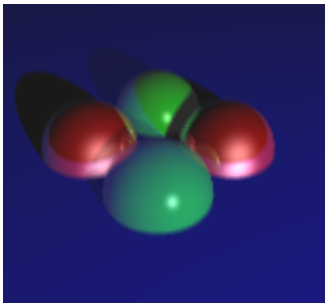


Figura 3.4.4 Imagen con oversampling 3 aplicado

Resultados

El alumno describirá los resultados obtenidos del desarrollo de su práctica.

Conclusiones

El alumno plasmará sus comentarios que demostrarán que cumplió el objetivo y reafirma el conocimiento adquirido.

Bibliografía:

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Kevin Suffern; Ray Tracing from the Ground Up; A K Peters/CRC Press (September 6, 2007)

3.5 PROYECTO II Raytracer Avanzado

Objetivo

Implementar funciones avanzadas a un raytracer básico como son: cálculo de sombras y fuentes de luz, sombreado e iluminación de superficies, reflexión, refracción y sobre muestreo.

Introducción

Al terminar el primer proyecto se completó un raytracer básico capaz de emitir rayos desde la cámara hacia la escena y revisar colisiones con los objetos en esta, obteniendo así su color del material el cual se consideró como el color final de la imagen. Sin embargo, no se pueden apreciar las características de ningún objeto en la escena y no se simula ningún tipo de iluminación. A lo largo de las últimas cuatro prácticas se introdujeron conceptos nuevos, como lo son las fuentes de luz y se fueron utilizando las propiedades del material que habían sido previamente definidas en la práctica 2, como son la componente difusa y especular de un material, así como el índice de refracción, entre otras.

Es importante mencionar que la mayor carga de código está en el primer proyecto, pues los conceptos básicos como calculo vectorial, color y trazado de rayos son esenciales para el funcionamiento correcto de nuestra aplicación y se siguen utilizando durante la realización de todas las prácticas posteriores. Sin embargo, la complejidad de conceptos y código en este proyecto y a lo largo de las prácticas del capítulo 3 es mucho mayor, aunque la cantidad de código a implementar es considerablemente menos.

Revisemos el material adicional que se agregó a nuestra implementación durante el capítulo 3, posteriormente se hará un resumen de cada práctica y se realizará una comparación de la evolución del proyecto hasta obtener la implementación final del programa.

Clase	Variables de clase	Métodos de clase
Light	type, position, color	
Scene	lights, oversampling, depth	
Raytracer		calculate_shadow

Tabla 3.5.1 – Listado de clases, variables y métodos nuevos

Desarrollo

Es importante probar la implementación de cada práctica para asegurarse de que el resultado es el deseado, para esto se recomienda utilizar la misma escena (o escenas) a lo largo del desarrollo de las prácticas del capítulo 3, ya que sólo se puede corroborar una correcta implementación de los algoritmos basándose en el resultado final de la imagen obtenida.

Se partirá del resultado final de la implementación básica y se utilizará la misma escena como referencia del avance obtenido. La figura 3.5.1 es el resultado final que se obtuvo en el Proyecto I y es la misma imagen que la figura 2.6.4, se incluye nuevamente como referencia y comparación.

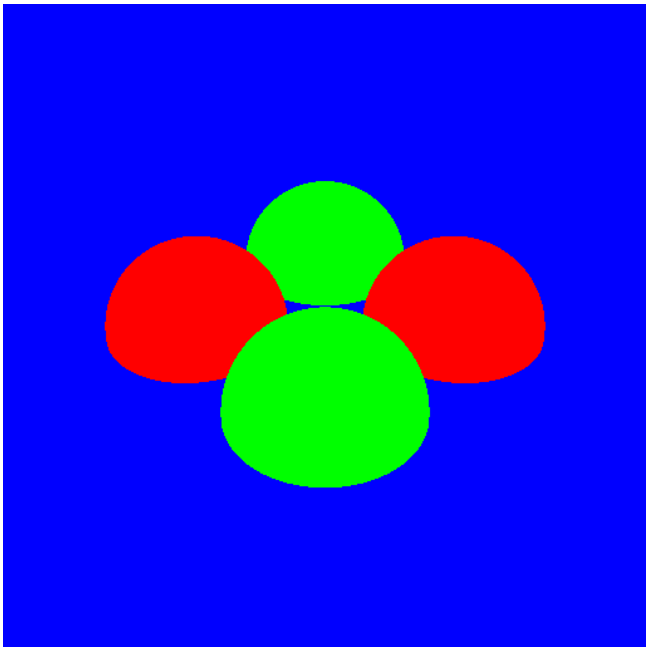


Figura 3.5.1 Ray tracing de una escena simple

En la práctica 6 se implementó la clase ***Light*** que define nuestras fuentes de luz, y se adecuó el código en la clase ***Scene*** para considerarlas dentro de la carga del archivo con la descripción de la escena. Posteriormente se creó un nuevo método ***calculate_shadow*** en la clase ***Raytracer*** de manera auxiliar para realizar el cálculo de las sombras creadas por estas fuentes de luz. Se agregaron dos fuentes de luz a la escena (ambas de color blanco), una de tipo ambiental (que simplemente aporta a toda la escena y se atenúa proporcionalmente) y otra de tipo puntual, que es la que genera sombras de objetos que se interponen entre el punto de colisión en línea directa con la fuente de luz. El código agregado al archivo de escena es el siguiente:

```
# light: type, position(x, y, z), color(r, g, b)

light ambient 0.0 0.0 0.0 0.1 0.1 0.1

light point 5.0 5.0 15.0 1.0 1.0 1.0
```

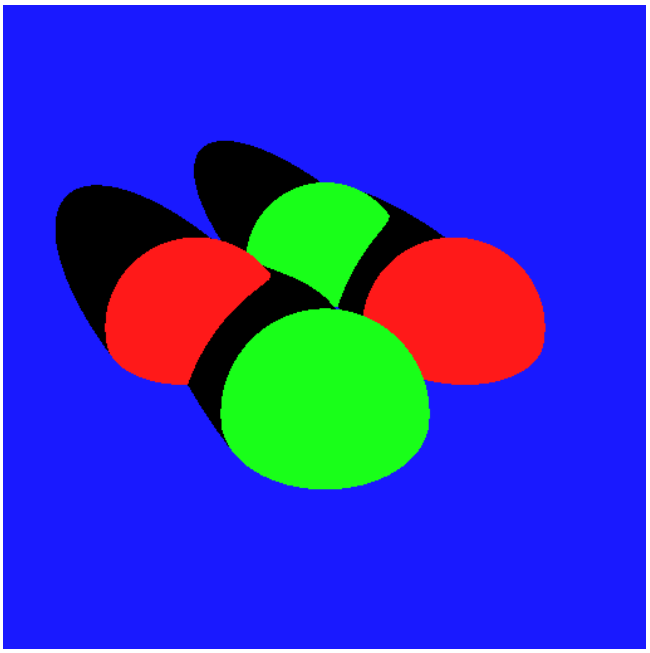


Figura 3.5.2 Cálculo de sombras

Se puede observar en la figura 3.5.2 el resultado correcto de la implementación, se recomienda mover la posición de la luz puntual para observar el comportamiento de las

sombras en la escena o bien agregar más fuentes de luz puntual para observar cómo interactúan múltiples luces en la generación de sombras. *Nota: La intensidad de la sombra es el valor* ***transmit*** *de nuestro material, pues es el que define la transparencia o transmitividad del material del objeto.*

Las prácticas 7 y 8 trabajan exclusivamente con el método ***trace*** de la clase ***Raytracer***, pues es donde se hace el cálculo de color y la simulación de la luz en la escena y se utilizan los rayos emitidos para poder generar nuevos rayos de reflexión y refracción, es aquí (práctica 8) donde se modifica el método ***trace*** para que tenga una naturaleza recursiva y se agrega el concepto de profundidad de trazado (***depth***) para poder detener el algoritmo.

En la práctica 7 se utiliza las componentes difusas y especulares, así como la intensidad de brillo del material para el cálculo del *shader* (sombreado) de la superficie. Esta implementación utiliza el sombreado de tipo *Phong,* como se menciona en la práctica, por ser uno de los más utilizados y simples de comprender, pero también por los excelentes resultados que se obtienen con él con un bajo costo computacional.

En esta práctica sólo se modifican los valores de ***diffuse, specular*** y ***shininess*** de los materiales para poder observar el resultado de sombreado de la implementación. Los valores modificados son los siguientes:

```
# material: color(r g b) diffuse specular shininess reflect transmit IOR

material 1.0 0.0 0.0 0.5 0.5 5.0 0.0 0.0 0.0

material 0.0 1.0 0.0 0.5 0.5 50.0 0.0 0.0 0.0

material 0.0 0.0 1.0 1.0 1.0 5.0 0.0 0.0 0.0
```

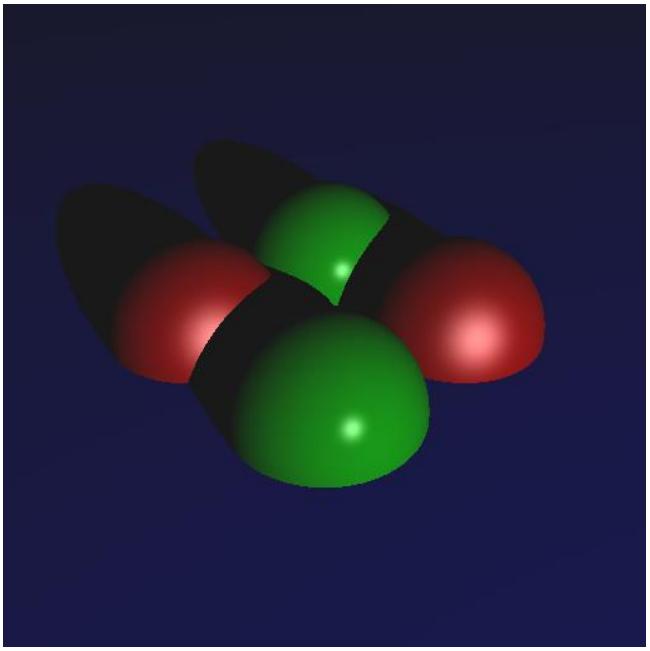


Figura 3.5.3 Phong Shading

En la figura 3.5.3 se puede observar el resultado de agregar sombreado a las superficies, se aprecia que la imagen se oscurece considerablemente, y es por eso que existen las fuentes de luz ambientales para iluminar la escena de manera general y se considera que el color base de los objetos no tiene ningún factor de iluminación (si no hay fuentes de luz todo se vería negro).

También se puede apreciar el cambio que tiene el valor de ***shininess*** en la superficie de las esferas, a mayor valor, el material se considera más “duro” y la luz no se propaga tanto en la superficie. Se recomienda hacer varias pruebas de valores difuso y especular para lograr la apariencia deseada de un material.

La práctica 8 implementa el algoritmo recursivo del cual se obtiene el nombre de trazado de rayos, pues dependiendo de las propiedades reflexivas o refractivas del material, se trazan nuevos rayos en el punto de colisión con la superficie y se sigue haciendo esto mientras se encuentren colisiones con materiales similares y no se haya alcanzado la profundidad máxima de trazado en el algoritmo. Para ejemplificar la implementación se modifiarán los valores de ***reflect, transmit*** e ***ior***, de los materiales de las esferas. El material rojo se hará parcialmente reflexivo y el verde parcialmente refractivo (ambos 0.0

< x < 1.0), para fines prácticos se modificará levemente el índice de refracción del material traslucido (verde) para verificar su funcionamiento, pero no lo suficiente para que no se pueda apreciar en la escena.

```
# material: color(r g b) diffuse specular shininess reflect transmit IOR

material 1.0 0.0 0.0 0.5 0.5 5.0 0.6 0.0 0.0

material 0.0 1.0 0.0 0.5 0.5 50.0 0.0 0.7 1.1
```

Los valores de ***reflect*** y **transmit** tienen un rango [0.0, 1.0] donde 0.0 significa que no hay reflexión (***reflect***) o que el objeto es completamente transparente (***transmit***), y un valor de 1.0 indica que es un espejo perfecto (***reflect***) o el objeto es completamente opaco (***transmit***), el índice de refracción depende de cada material que queremos simular y el valor de 1.0 significa que la luz no se refracta en el medio.

De igual manera hay que agregar si se incluye la propiedad ***depth*** en el archivo de descripción de escena, esta será la profundidad de trazado del algoritmo, si se ignora tiene un valor predeterminado de 3, es un valor entero positivo.

En la figura 3.5.4 se aprecia la reflexión en las esferas rojas y la transparencia en las esferas verdes, también se puede ver la distorsión de la esfera roja a través de la verde debido a la refracción de la luz en el material. Es importante también notar que la sombra de las esferas verdes se percibe de manera más tenue pues como se mencionó anteriormente, se utiliza la propiedad ***transmit*** como factor de atenuación al generar las sombras provocadas por estos objetos. Se recomienda modificar los valores para percibir las diferencias, así como mezclar ambas propiedades, pues objetos traslucidos como el vidrio también tienden a ser reflexivos, para los índices de refracción de cada material se recomienda consultar una tabla de propiedades del material en cuestión (como la incluida en la práctica 8), ya que varían considerablemente, se puede tomar como ejemplo el del agua que es de 1.333 o el del vidrio que puede variar entre 1.44 y 1.9 dependiendo de su grosor y composición.

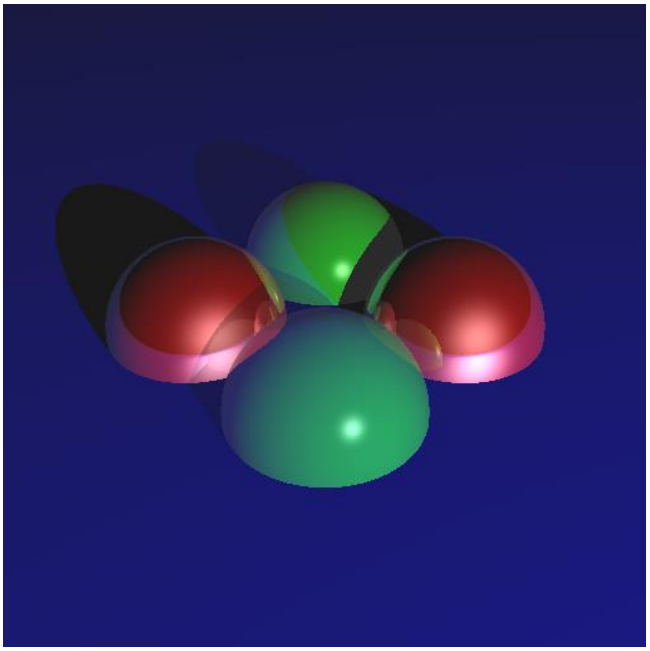


Figura 3.5.4 Reflexión y refracción

La práctica 9 modifica exclusivamente el método ***trace_pixel*** de la clase ***Raytracer*** creando una sub cuadricula por cada pixel de la imagen trazando rayos y promediando el resultado para obtener bordes más suaves en la imagen final, a este concepto se le llama *Anti-Aliasing* y es comúnmente utilizado en la computación gráfica. El costo computacional del *oversampling* (sobre muestreo) es muy notorio en el tiempo de ejecución del programa y debe de usarse únicamente cuando se está seguro de que el resultado final será el adecuado (después de varias pruebas de menor calidad), pues la cantidad de rayos trazados escala cuadráticamente.

El único valor que se requiere agregar al archivo de escena es el de ***oversampling*** que tiene un valor predeterminado de 1, lo que significa que no hay sobre muestreo, ya que se generaría una sub cuadricula de 1x1 por cada pixel, si se incrementa el valor a 2, la sub cuadricula generada es de 2x2 lo que incrementa en 4 veces la cantidad de rayos trazados, un valor de 3 lo incrementa en 9 y así subsecuentemente para valores más grandes. Se hace notar que valores superiores a 4 tienen muy poco impacto en el resultado final de la imagen y deben de ser evitados.

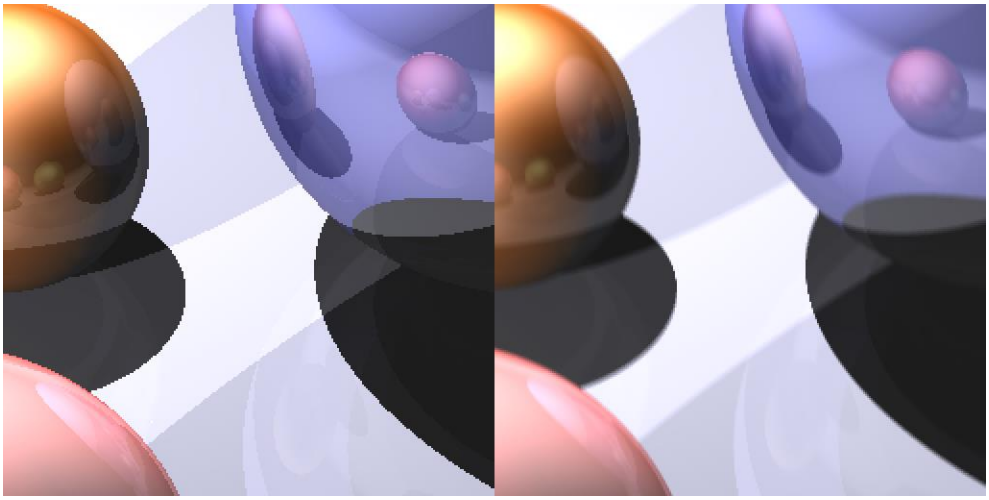


Figura 3.5.5 Oversampling

En la figura 3.5.5 se puede apreciar la diferencia que hace un valor predeterminado de 1 (izquierda) y un valor de 4 (derecha) en la calidad de los bordes de la imagen. Siempre puede incrementarse la resolución de la imagen dependiendo del resultado que se requiera obtener. La imagen anterior fue generada con la referencia de la implementación final (figura 3.5.6), a continuación, se incluye el contenido del archivo de descripción de escena con el que fue generada.

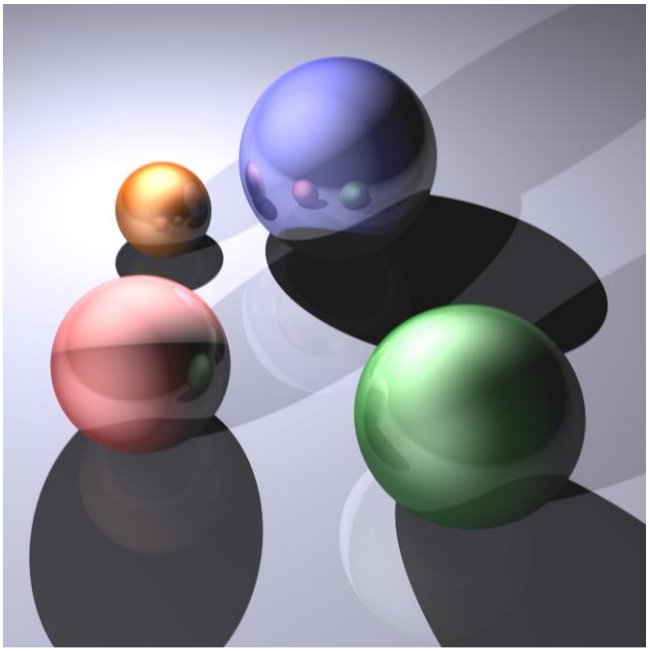


Figura 3.5.6 Implementación final de referencia

```
# Archivo de referencia Proyecto 2
image_size 640 640
depth 5
oversampling 2
field_of_view 60

camera_position 5.0 5.0 5.0
camera_look 1.0 1.0 0.0
camera_up 0.0 0.0 1.0

# sphere: material x y z radius
sphere 3 3.0 1.5 0.8 0.8
sphere 4 0.0 0.0 1.0 1.0
sphere 2 1.0 3.0 0.7 0.7
sphere 0 -1.0 1.5 0.5 0.5

# plane: material normal_x normal_y normal_z distancia:
plane 1  0.0 0.0 1.0  0.0

# light: type x y z  r g b
light point -2.0 1.0 4.0 1.0 1.0 1.0
light point 2.0 13.0 4.0 0.5 0.5 0.5
light ambient 0.0 0.0 0.0 0.1 0.1 0.1

# material: r g b diffuse specular shininess reflect transmit ior
material 1.0 0.4 0.0 0.9 0.8 2.0 0.15 0.0 0.0
material 0.8 0.8 0.9 1.0 0.0 4.0 0.1 0.0 0.0
material 1.0 0.2 0.2 0.9 0.7 1.0 0.2 0.0 0.0
material 0.2 1.0 0.2 0.4 0.8 2.0 0.15 0.0 1.1
material 0.3 0.3 1.0 0.7 0.4 0.8 0.3 0.0 0.0
```

Se puede encontrar una implementación de referencia en su última versión en:

https://git.io/vDUAu

4 CAPÍTULO IV TEMAS ADICIONALES Y ESPECIALES

4.1 Malla de triángulos

Introducción

Triángulo

Un triángulo es una primitiva geométrica que está definido por tres vértices (o puntos) cuyas posiciones están definidas en el espacio tridimensional y los cuales definen un plano. La normal es perpendicular a este plano (ver Figura 4.1.1)

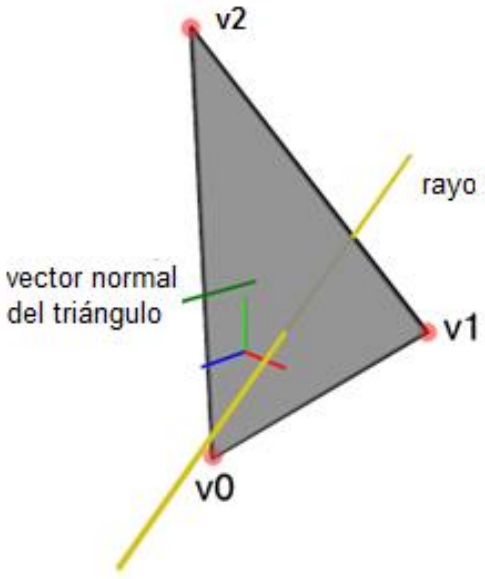


Figura 4.1.1 Geometría de un triángulo. Un rayo (amarillo) intersecta este triángulo[37]

Un solo punto puede representar una ubicación. Con dos puntos se puede definir una línea. Con tres puntos, se puede definir una superficie (un plano). Un triángulo es coplanar, es decir, los tres vértices definen un plano y éstos se encuentran en el mismo. Esto no pasa con los polígonos de más de tres vértices; cuatro puntos definen una región cuádruple incluso cuando dichos puntos no están en la misma superficie. Entonces, ésta se puede convertir en dos triángulos coplanares tal como se muestra en la Figura 4.1.2.

[37] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

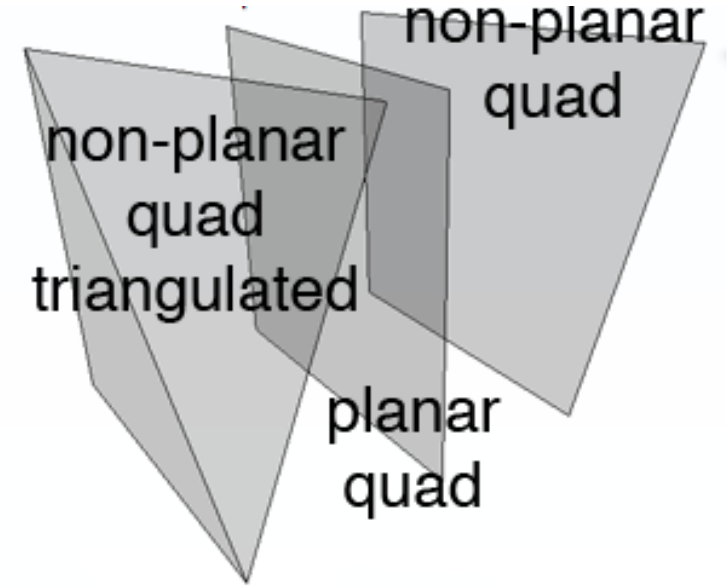


Figura 4.1.2 Una región cuádruple no necesariamente es coplanar[38]

Esta técnica puede ser aplicada a un polígono que tenga un determinado número de vértices: este proceso es llamado triangulación.

¿Cómo se calcula la intersección de un rayo con un triángulo?

Por décadas se han desarrollado múltiples algoritmos para calcular la intersección entre un rayo y un triángulo.

Para empezar a explicar dicha intersección, se parte de las siguientes preguntas:

- El rayo intersecta el plano definido por el triángulo?
- Los puntos de intersección se encuentran dentro del triángulo?

Herramientas matemáticas básicas

Los vértices del triángulo definen un plano. Cada uno de ellos tiene un vector normal el cual puede ser visto como una línea perpendicular a la superficie del plano. Considerando que se saben las coordenadas de los vértices (A, B y C), se pueden calcular los vectores

[38] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

AB y AC (los cuales son simplemente las líneas que van de A a B y A a C) Estos dos vectores se encuentran en el mismo plano.
Para calcular la normal al plano, simplemente se obtiene el producto cruz de AB y AC. El vector que resulta (denotado N) es el normal que se observa en la Figura 4.1.1.

Estas operaciones se muestran en el pseudocódigo ejemplo:

Vec3f v0(-1, -1, 0), v1(1, -1, 0), v2(0, 1, 0);

Vec3f A = v1 - v0; // lado 0

Vec3f B = v2 - v0; // lado 1

Vec3f C = cross(A, B); // este vector es el normal al triángulo.

normalize(C);

Si se normaliza C, se obtiene el vector (0,0,1) el cual es paralelo al eje z positivo que es lo que se esperaba ya que los vértices v0, v1 y v2 se encuentran en el plano xy.

Intersección rayo-triángulo

Ahora que se entiende como calcular el vector normal al triángulo, se necesita encontrar la posición del punto P, que es el punto donde el rayo intersecta el plano.
Paso 1: Encontrar el punto P
Se sabe que P esta en alguna lugar del rayo definido según su origen O y su dirección R. La ecuación paramétrica del rayo es: $P = O + tR$ donde t es la distancia del origen del rayo O a P. Para encontrar P se necesita encontrar t (figura 1). La ecuación del plano es : $Ax + By + Cz + D = 0$, donde A, B y C pueden ser vistos como las coordenadas de la normal al plano y D es la distancia del origen (0,0,0) al plano. Las variables x, y y z son las coordenadas de algún punto que se encuentra en el plano.
Se sabe que tanto el plano normal como los tres vértices del triángulo (V0, V1, V2) se encuentran en el plano. Entonces es posible calcular D. Alguno de los tres vértices puede ser elegido pero se usará el V0:
Usando pseudocódigo:
float D = dotProduct(N, v0);

```
// también se puede calcular el producto punto directamente
float D = N.x * v0.x + N.y * v0.y + N.z * v0.z;
```

También se sabe que el punto P el cual es el punto de intersección entre el rayo y el plano, se encuentra en el plano. Entonces, se puede sustituir P por (x,y,z) como sigue:

$$P = O + tR$$

$$Ax + By + Cz + D = 0$$

$$A * Px + B * Py + C * Pz + D = 0$$

$$A * (Ox + tRx) + B * (Oy + tRy) + C * (Oz + tRz) + D = 0$$

Finalmente:

$$t = -\frac{N(A,B,C)\cdot O+D}{N(A,B,C)\cdot R} \quad \text{(Ecuación 1)}$$

Se debe tener cuidado cuando se implemente esta técnica. De entrada, la cámara está orientada a lo largo del eje z negativo, así todos los rayos primarios tienen coordenadas z negativas. Esto también significa que cuando la normal del triángulo esta frente a la cámara, el producto punto de la normal con los rayos primarios será negativo. Debido a eso, el signo negativo de la ecuación será cancelado por el signo negativo del denominador.

$$t = -\frac{N(A,B,C)\cdot O + D}{-N(A,B,C)\cdot R}$$

$$t = \frac{N(A,B,C)\cdot O + D}{N(A,B,C)\cdot R}$$

El rayo y el triángulo son paralelos

Si el rayo y el plano son paralelos, no intersectarán (Figura 4.1.3). Esto quiere decir que si el triángulo es paralelo a la dirección del rayo, entonces, la normal del triángulo y la dirección del rayo son perpendiculares.

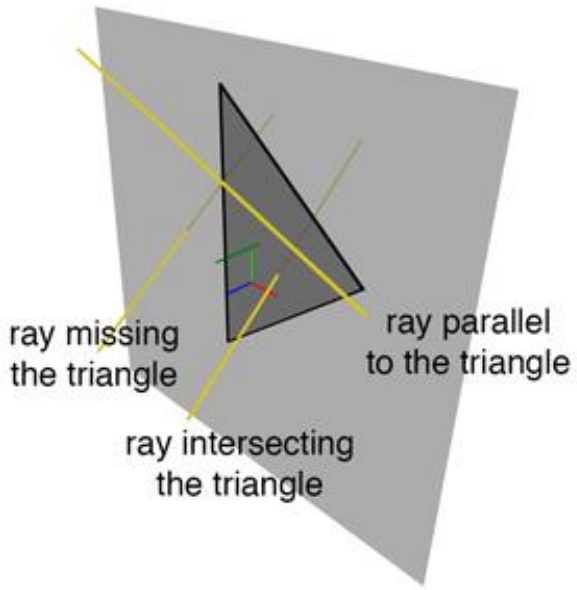


Figura 4.1.3 Si el rayo es paralelo al triángulo entonces no hay intersección[39]

Se recomienda que antes de calcular t, se calcule el resultado de $N \cdot R$ primero y si el resultado es 0, entonces no hay intersección.

El triángulo esta detrás del rayo

Hasta este momento se ha asumido que el triángulo esta siempre frente al rayo. Pero que pasa si el triángulo esta detrás del rayo con el rayo aún viendo hacia la misma dirección. Normalmente el triángulo no debería ser visible. En este caso, t (ecuación 1) es negativo, lo que hace que el punto de intersección esté en la dirección opuesta que la dirección del rayo. En resumen, si t es menor que 0, el triángulo esta detrás del origen del rayo y no es visible. No hay intersección. Si t es mayor que 0, el triángulo es visible por el rayo.

[39] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

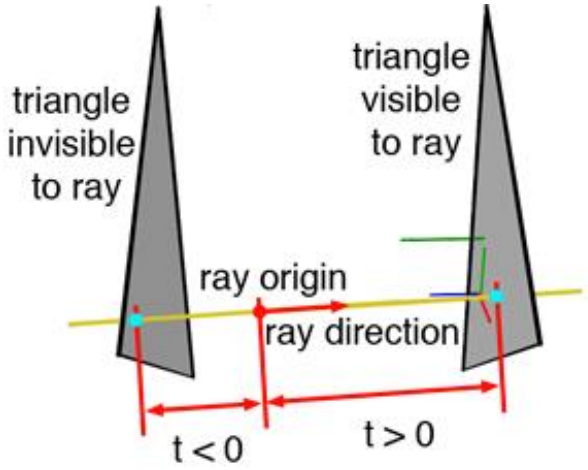


Figura 4.1.4 Si un triángulo esta "detrás" del rayo, entonces el rayo no debería ser visible[40]

Paso 2: P esta dentro o fuera del triángulo.

Ahora que se ha encontrado el punto P el cual es el punto donde el rayo y el plano intersectan, se tiene que definir si P esta dentro del triángulo (en este caso el rayo intersecta el triángulo) o si P esta fuera (en este caso el rayo no intersecta el triángulo). La Figura 4.1.4 ilustra estos casos.

La solución a este problema es simple y es llamado la prueba dentro-fuera. Imagine un vector A el cual esta alineado con el eje x (Figura 4.1.5). Imagine que este vector esta actualmente alineado con un borde del triángulo (el borde definido por los dos vértices V0V1). Ahora el segundo borde B, esta definido por los vértices V0 y V2 como se muestra en la Figura 4.1.5. Calcule el producto cruz de esos dos vectores. Como era de esperarse, el resultado es un vector el cual esta apuntando en la misma dirección que el eje z y el vector normal del triángulo.

[40] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

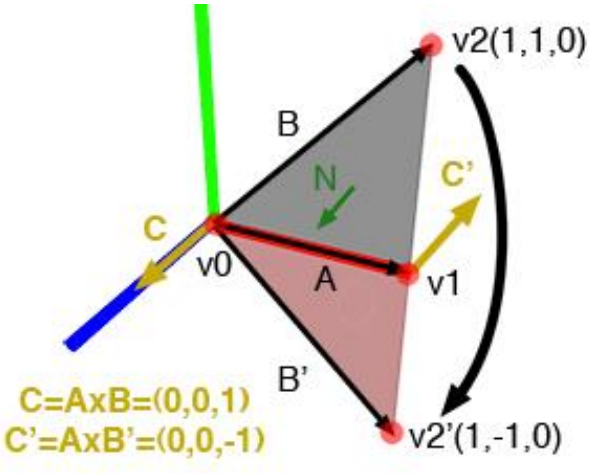


Figura 4.1.5 El punto C y C' en direcciones opuestas[41]

Ahora imagine que en lugar de tener las coordenadas (1,1,0) el vértice V2 tiene las coordenadas (1,-1,0). En otras palabras, se tiene un espejo de esta posición en el eje x. si se calcula el producto cruz AxB' se obtiene el resultado $C' = (0,0,-1)$
Como C y N están apuntando en la misma dirección, su producto punto da un valor mayor que 0 (positivo). Ahora, como C' y N están apuntando en direcciones opuestas, su producto punto da un valor menor que 0 (negativo).

Algortimo Möller-Trumbore

El algoritmo Möller-Trumbore (o MT) es un algoritmo rápido de intersección rayo-triángulo el cual fue introducido en 1997 por Tomas Möller y Ben Trumbore. Hoy en día es considerado un algoritmo rápido el cual es usado como punto de referencia para comparar el resultado de otros métodos aunque una comparación justa del algoritmo de intersección rayo-triángulo es actualmente algo difícil de hacer porque la rapidez depende de muchos factores tal como la manera en que los algoritmos estén implementados, el tipo de escena prueba que se utiliza, si los valores son precalculados, etc.

[41] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

El algoritmo MT aprovecha la parametrización de P, el punto de intersección en términos de coordenadas baricéntricas. Dicho punto se calcula usando la siguiente ecuación:

$$P = wA + uB + vC$$

Si $w = 1 - u - v$

$$P = (1 - u - v)A + uB + vC$$

Desarrollando la fórmula, se obtiene lo siguiente:

$$P = A - uA - vA + uB + vC = A + u(B - A) + v(C - A)$$ (ecuación 2)

Note que (B-A) y (C-A) son los lados AB y AC en el triángulo ABC. La intersección P puede ser escrita usando la ecuación paramétrica del rayo:

$$P = O + tD$$ (ecuación 3)

Donde t es la distancia del origen del rayo a la intersección P. Si se remplaza P (ecuación 3) en la ecuación 2, obtenemos:

$$O - A = -tD + u(B - A) + v(C - A)$$

En el lado izquierdo de la ecuación, se tienen tres valores desconocidos (t, u, v) multiplicados por tres valores conocidos (B-A, C-A, D). Se puede reacomodar la ecuación usando la siguiente notación:

$$[-D \quad (B - A) \quad (C - A)] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A$$ (ecuación 4)

Imagine que tiene un punto P dentro del triángulo ABC. Si se transforma el triángulo de varias maneras, por ejemplo la traslación, rotación, etc., las coordenadas del punto P (x, y, z) cambiarían. Por el contrario, si se expresan las coordenadas del punto P en coordenadas baricéntricas, las transformaciones aplicadas al triángulo no afectan dichas coordenadas del punto (ver Figura 4.1.6)

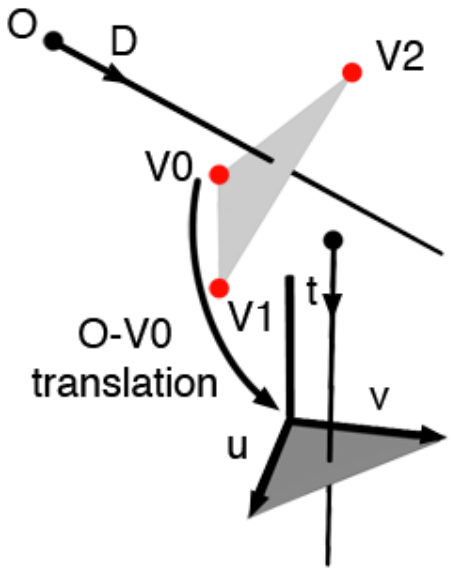


Figura 4.1.6 Se pueden expresar la posición de P en el espacio t, u y v[42]

El algoritmo MT aprovecha esta propiedad. Definir un nuevo sistema coordenado donde las coordenadas de P no están definidas en términos de x, y y z, sino en términos de u y v. Las coordenadas u y v no pueden ser mayor que 1 o menor que 0. Su suma no puede ser mayor que 1 (u+v<=1). Éstas expresan coordenadas de puntos definidos dentro de un triángulo unitario (este es el triángulo definido en el espacio u,v con vértices (0,0), (1,0), (0,1) como se muestra en la Figura 4.1.7).

Resumiendo, se ha reinterpretado la posición del punto P (x,y,z) en términos de coordenadas baricéntricas u y v, esto es, del espacio xyz al espacio uv.

La ecuación 4 tiene tres valores no conocidos: u, v y t. Geométricamente, se ha explicado el significado de u y v. Se debe considerar que t es el tercer eje del sistema de coordenadas u y v. La variable t define la distancia del origen del rayo a P, el punto de intersección.

[42] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

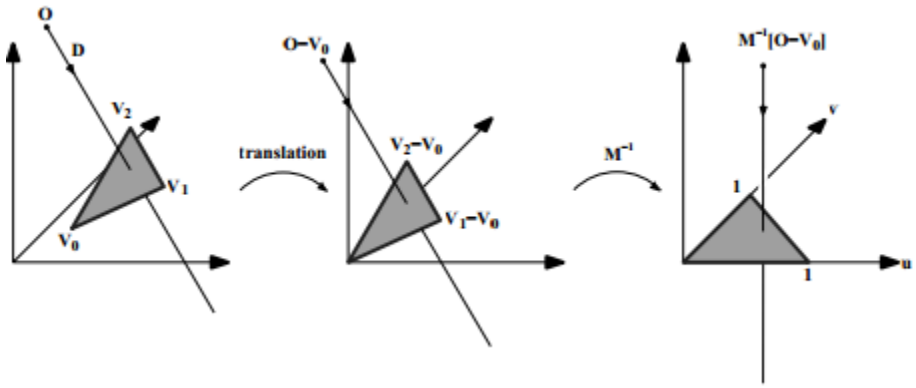


Figura 4.1.7 Cambio de coordenadas baricéntricas[43]

Mallas triangulares

Una malla triangular es un tipo de malla poligonal que comprende un conjunto de triángulos, en nuestro caso en tres dimensiones, que se conectan por sus lados o vértices comunes y aproximan una superficie. Aunque el campo de aplicación de la generación automática de mallas triangulares ha sido tradicionalmente la obtención de modelos digitales de elevaciones del terreno, hoy en día son usadas en renderización, animación, realidad virtual y en análisis de elementos finitos.

Elementos y representación

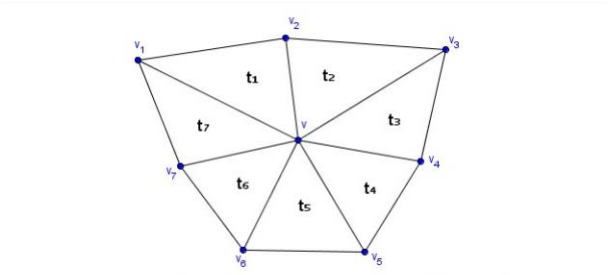
Sea Ω una malla triangular. Los triángulos t_i de la malla cumplen:

i. Ω = $U_i t_i$
ii. Cada triangulo es cerrado y con interior no vacío.
iii. Para diferentes triángulos t_i y t_j la intersección de sus interiores es vacía.
iv. Para diferentes triángulos t_i y t_j su intersección es un lado común, un vértice común o es vacía.

La mayoría de mallas triangulares almacenadas en la red o construidas por generadores de malla como Gmsh, son mallas triángulo-vértice, FV (Face-Vertex) por sus siglas en

[43] https://www.lanshor.com/interseccion-rayo-triangulo/

inglés, es decir, mallas triangulares que se representan por la conectividad de sus triángulos y sus vértices. Una configuración como la mostrada en la Figura 4.1.8 se denomina vecindad, siguiendo el uso común de la topología dentro de un contexto discreto. Así, la vecindad de un vértice v_N es la colección de triángulos incidentes y vértices adyacentes a este. Los triángulos incidentes a v_N son aquellos triángulos que lo usan como uno de sus vértices, mientras los vértices adyacentes a v_N son los vértices distintos a este que son vértices de los triángulos incidentes. Por ejemplo, en la Figura 4.1.7, v1, v3, . . . , v7 son los vértices adyacentes a v mientras t1, t2, . . . , t7 son los triángulos incidentes. La representación FV de una malla triangular usa dos matrices. Una de ellas, denominada lista de vértices, guarda las coordenadas de los vértices, y la otra, llamada lista de triángulos, almacena la definición de los triángulos de acuerdo a sus vértices. En la Figura 4.1.8 se muestra la representación FV de la vecindad de v.



Lista de vértices

v	-0.0709, 0.1498, 0.0389
v_1	-0.0707, 0.1535, 0.0329
v_2	-0.0646, 0.1534, 0.0329
v_3	-0.0650, 0.1485, 0.0383
v_4	-0.0683, 0.1457, 0.0421
v_5	-0.0725, 0.1463, 0.0430
v_6	-0.0753, 0.1490, 0.0396
v_7	-0.0762, 0.1531, 0.0322

Lista de triángulos

t_1	$v\ v_1\ v_2$
t_2	$v_3\ v_2\ v$
t_3	$v\ v_3\ v_4$
t_4	$v\ v_4\ v_5$
t_5	$v\ v_5\ v_6$
t_6	$v\ v_6\ v_7$
t_7	$v\ v_1\ v_7$

Figura 4.1.8 Vecindad de v, lista de vértices adyacentes y lista de triángulos incidentes[44]

[44] http://www.bdigital.unal.edu.co/9424/1/1077446730.2013.pdf

Plano Promedio Uno de los elementos asociados a los vértices de una malla triangular es el plano promedio. Dado que tres puntos en el espacio definen un plano, entonces cada triángulo de la malla tiene un plano asociado. Así, para el vértice v hay un plano por cada triángulo incidente. El plano promedio es entonces el "promedio" de estos planos asociados.

Las mallas triangulares, son una de las técnicas más utilizadas en la generación de modelos computacionales de superficies de objetos 3D; aunque el proceso de generación de éstas, implica tratar con dos problemas: la determinación de la conectividad entre los puntos de datos, y el costo computacional tanto del procesamiento de los mismos, como del almacenamiento de su representación.

Es importante representarlos de manera eficiente para:

a) Reducir el espacio de almacenamiento.
b) Reducir el consumo de ancho de banda.
c) Reducir el tiempo de dibujado.

Tipos de representación:

Caras independientes.

Cada cara almacena sus vértices.

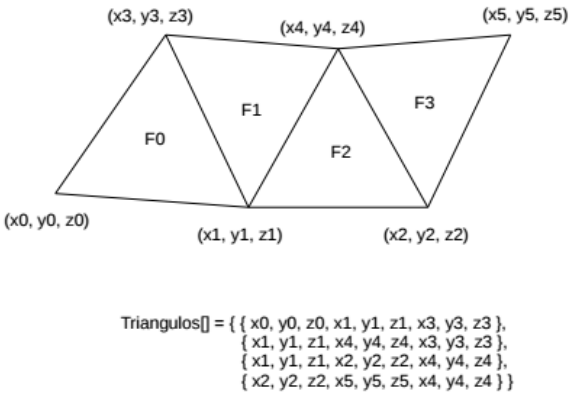


Figura 4.1.9 Caras independientes[45]

[45] http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf

Vértices compartidos

Cada cara almacena índices a una lista de vértices.

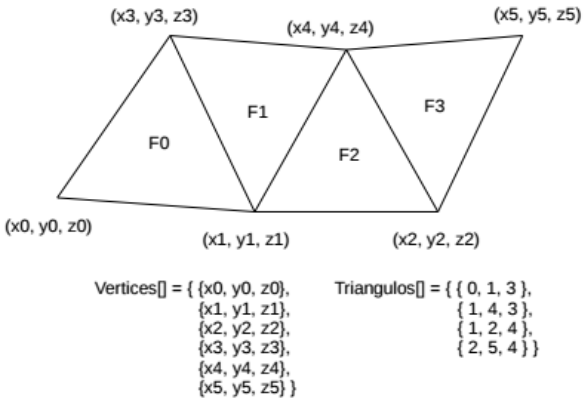


Figura 4.1.10 Vértices compartidos[46]

Tiras y abanicos de triángulos

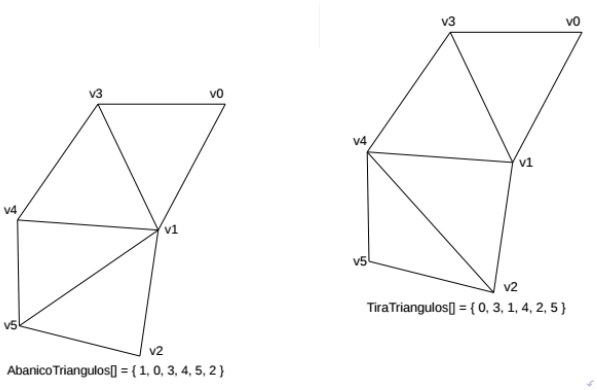


Figura 4.1.11 Tiras y abanicos de triángulos[47]

[46] http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf

[47] http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf

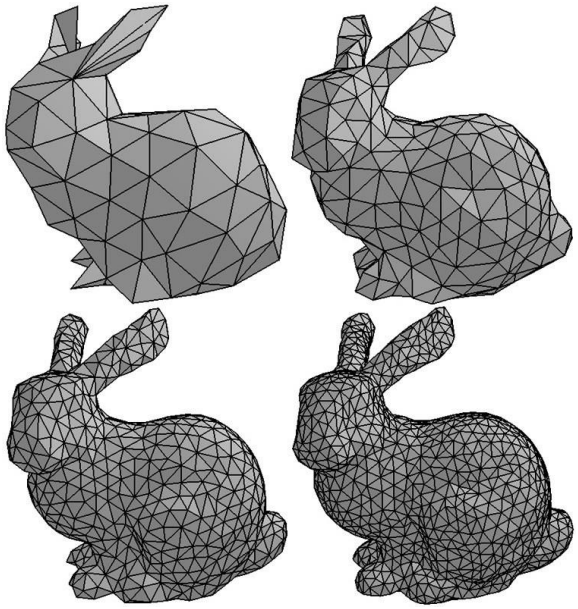


Figura 4.1.12 Modelado con malla de triángulos[48]

Bibliografía

Fuentes online:

https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

https://www.lanshor.com/interseccion-rayo-triangulo/

http://www.bdigital.unal.edu.co/9424/1/1077446730.2013.pdf

http://publicaciones.unisimonbolivar.edu.co/rdigital/inovacioning/index.php/identic/article/viewFile/20/28

http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf

https://www.quora.com/What-is-a-mesh-in-OpenGL

[48] http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf

4.2 Texturizado

Introducción

Una Textura es un arreglo de datos que contiene información de color y transparencia de un objeto que se encuentra dentro de nuestra escena, también puede contener información de: normales, profundidades y sombras, se aplican a nivel pixel, cada uno de los elementos del arreglo de datos que conforman una textura, reciben el nombre de ***Texel (Texture Element o Texture Pixel)*** Elemento de textura, es importante resaltar que pixel y texel no son lo mismo, ya que un pixel es la unidad más pequeña que se puede representar en nuestra escena y texel es un arreglo con patrones de iluminación los cuales pueden ser muy pequeños o grandes, dependiendo del tipo de textura que quiera aplicar.

Existen dos clasificaciones para determinar los tipos de texturas:

La primera se basa por la forma en que son generadas las texturas de esta clasificación existen dos tipos de texturas:

Las texturas de archivos Almacenados en Memoria. Son Texturas que se encuentran en archivos de imágenes (.bmp, .jpg, .tga, .raw) en un medio de almacenamiento, es decir, son imágenes que se aplicaran a nuestros objetos para tomen los colores de dicha imagen sin tener que indicarlo mediante un cálculo a cada pixel, lo que nos ahorra tiempo computacional ya que no se realiza el cálculo de rayos en toda la escena, sino que el objeto tomara los colores de la imagen que le proporcionemos acorde a las coordenadas del objeto y la imagen, por lo que es importante definir bien los límites de la textura, de lo contrario los demás objetos de la escena también se verán perjudicados por la aplicación de la textura (ver figura 4.2.1).

Figura 4.2.1 Texturas en formato jpg[49]

[49] http://www.freepik.es/foto-gratis/textura-acolchada-negra_871291.htm Freepik

Texturas Procedimentales/Procedurales: Son Texturas que son generadas mediante la evaluación de una función y/o algoritmo, el cual da cómo resultado valores de color que corresponden a los Texeles del objeto que se le aplicara la textura, este caso se requiere de mayor trabajo computacional ya que al aplicar un algoritmo se tienen que hacer mayor número de operaciones para definir las áreas de los objetos que se iluminaran y también para definir el patrón del comportamiento del texel (ver figuras 4.2.2 y 4.2.3).

Figura 4.2.2 Esfera con diferentes texeles aplicados[50]

Figura 4.2.3 Figura con texturas procedurales en los ladrillos del techo[51]

El segundo tipo está basado en el tamaño de las texturas

De esta clasificación existen tres tipos de texturas:

[50] http://www.oocities.org/valcoey/texturas_procedurales.html Oocities

[51] https://parpatrimonioytecnologia.wordpress.com/2016/06/13/tutorial-como-texturizar-de-forma-procedural-texture-painting-un-tejado-en-blender-cycles/ PAR

Texturas de Una Dimensión (1D). Son texturas que solo tienen un Texel de altura y/o un Texel de ancho, es decir, nos proporciona el efecto de ver los objetos de nuestra escena como si estuviéramos de frente a ellos y tuvieran un corte plano, todos por igual, lo que no es muy útil, ya que en el diseño asistido por computadora nuestro objetivo principal es obtener el mayor realismo posible (ver figura 4.2.2.4).

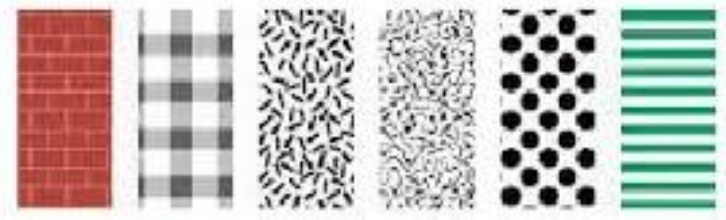


Figura 4.2.4 Textura de una dimensión[52]

Texturas de Dos Dimensiones (2D). Son texturas que tienen más de un Texel de altura y de ancho, por lo que podremos generar efectos de movimiento y profundidad con una única textura, a lo que se le puede generar mayor provecho ya se puede generar efectos que simulan acercamiento y lejanía (ver figura 4.2.2.5).

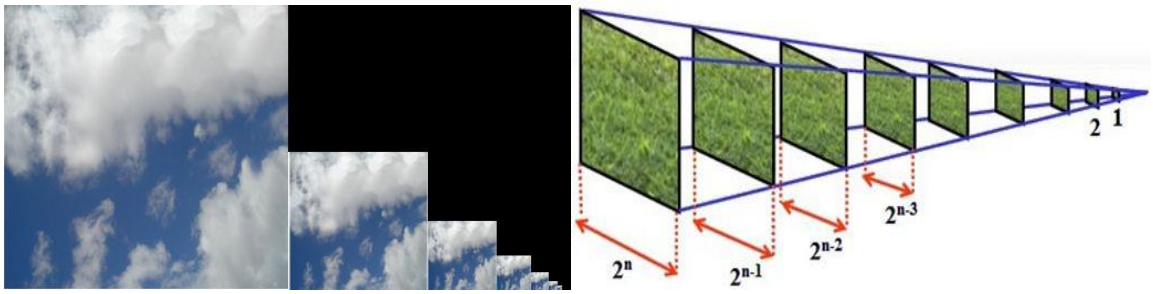


Figura 4.2.5 Figura de dos dimensiones[53]

Texturas de Tres Dimensiones (3D). Son texturas que tienen la capacidad de generar la sensación de volumen a los objetos que se muestran en las figuras, en la aplicación de este tipo de texturas no se puede hacer la asignación directa de un texel a un pixel, debido a los siguientes factores:

[52] http://www.mailxmail.com/curso-nociones-basicas-diseno/texturas-escalas-dimensiones Mailxmail
[53] http://es.slideshare.net/mpazmv/tema-2-puntolnea-plano-textura Slider Share

a) La posición de la superficie puede cambiar durante la ejecución del programa, por lo tanto también cambian los texel asignados.

b) El tamaño de una textura NO siempre corresponde al tamaño de la superficie a la cual será aplicada.

Es por ello que se ocupa el espacio de textura, es decir, la textura está definida mediante coordenadas, las cuales serán asignadas a las coordenadas cartesianas de los vértices de la superficie, el espacio de textura, define las coordenadas que serán asignadas a los vértices de la superficie, no importa el tamaño de la textura.

Generalmente se utilizan los planos (s, t) u (u, v) para el trabajo con este tipo de texturas, las coordenadas que definirán a la textura (s, t) o (u, v) siempre estarán entre los valores [0, 1] es por ello que, generalmente, se trabaja con texturas rectangulares, para facilitar la obtención de las coordenadas en el espacio de textura.

Algo muy importante a recalcar es que, aun cuando nuestra textura estará definida entre valores de [0, 1], las coordenadas que se asignan a los vértices, pueden o no estar entre dichos valores, estas últimas coordenadas, que reciben el nombre de coordenadas de texturizado, son las coordenadas finales que reciben los vértices de la superficie para la asignación de los texeles de la textura.

Aunque esta técnica de texturizado es la más empleada en nuestros días, en nuestro tema central de trabajo de tesis es solo un complemento ya que la generación de colisiones y colores principalmente la obtenemos del comportamiento del algoritmo Raytracer, aunque podemos aplicar algunas texturas en objetos grandes que sean de color uniforme, solo como una herramienta complementaria no esencial (ver figura 4.2.6 y figura 4.2.7).

Figura 4.2.6 Textura con tres dimensiones

Figura 4.2.7 Textura con tres dimensiones[54]

Filtros

Los filtros son algoritmos que se aplican a las texturas, cuando éstas tienen que ser redimensionadas para poder ser aplicadas a una superficie, en aplicaciones de computación gráfica en tiempo real, los objetos están constantemente cambiando de tamaño, por lo cual la utilización de filtros es continua.

Debe existir un compromiso entre la calidad y el tiempo de respuesta de un filtro.

En general, los pasos a seguir para realizar el Mapeado de Texturas, son:

1. Habilitar y configurar el Mapeo de Texturas.
2. “Crear” una textura.

[54] http://es.aliexpress.com/store/product/Red-brick-Vintage-three-dimensional-brick-wallpaper-roll-natural-stone-brick-effect-home background-decor/915595_32260275019.html?ws_ab_test=201526_2,201527_4_71_72_73_74_75,201409_2 Aliexpress

3. Especificar la superficie a la cual se aplicará.
4. Indicar cómo se aplicará la textura a cada pixel mediante las Coordenadas de Texturizado.
5. Mapeo de Texturas.

   La configuración consiste en determinar el comportamiento de la textura con Coordenadas de Texturizado mayores a 1 y/o menores a 0:

6. Crear una Textura.

   No hay funciones para "crear" texturas, es tarea del diseñador el incorporar el algoritmo de creación (Procedimentales) o el lector de archivos de imágenes (Almacenadas) para obtener la información de color.

   (tipo, #mipmaps, formato, ancho, alto, borde, formato datos, tipo datos, datos**);**

   Especificar la superficie a la cual se aplicará.

   Se utiliza la función:

   (tipo, indiceTextura **);**

   Las definiciones de superficies que se encuentren después de la función se verán afectadas por la textura, siempre y cuando tengan Coordenadas de Texturizado.

   indiceTextura = 0, indica que no se ocupará textura.

7. Coordenadas de Texturizado.

   Las superficies que se deseen dibujar con una textura, deberán tener asignados a sus vértices, sus correspondientes Coordenadas de Texturizado.

Por otra parte, los filtros más utilizados son:

- NEAREST.
- LINEAR.
- NEAREST_MIPMAP_NEAREST.
- LINEAR_MIPMAP_NEAREST.

- NEAREST_MIPMAP_LINEAR.
- LINEAR_MIPMAP_LINEAR.

En la actualidad las texturas son de gran utilidad ya que permiten ahorrar trabajo computacional en el diseño asistido por computadora, dentro de las escenas que se diseñan siempre existen áreas u objetos que su comportamiento es estático, dicho en otras palabras no cambia ni de posición ni de color, lo que nos resulta más fácil emplear una textura, que realizar cálculos innecesarios de manera repetida.
La implementación de las herramientas de desarrollo que referimos en este capítulo son las más sencillas de implementar en el diseño de gráficos en computadora, ya que son instrucciones sencillas a las que les damos únicamente valores, para que el programa realice los cálculos necesarios y nos muestre los resultados en la escena que se diseña, dependiendo de lo que estamos representado será el tipo de textura que utilizamos aunque la más utilizadas don de tres dimensiones ya que nos ayudan simular escenas de mayor realismo al contener volumen.

Bibliografía:

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014
FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Kevin Suffern; Ray Tracing from the Ground Up; A K Peters/CRC Press (September 6, 2007)

4.3 Procesamiento Paralelo

Introducción

El algoritmo de raytracing es un proceso costoso en procesamiento, se ha podido comprobar que conforme se agregan más características al programa, el proceso para obtener nuestra imagen final se vuelve más lento. Esto es cierto para cualquier simulación, si buscamos un mejor resultado, más apegado a la realidad, es necesario realizar más cálculos para obtenerlo. Sin embargo, en el caso del raytracing, la naturaleza del algoritmo lo vuelve exponencialmente más lento conforme se agregan cálculos a la función de trazado, que es la función principal que debe repetirse millones de veces para obtener el color final de la imagen en cada instancia. Al mismo tiempo, la complejidad de la escena que se quiera simular agrega proporcionalmente tiempo de procesado a nuestro resultado final, mientras que optimizaciones en el código de colisión o bien la subdivisión espacial de la escena por medio de estructuras de datos y arboles pueden ayudar en ese ámbito, no son una solución general al problema del procesamiento en el trazado de rayos.

Ahora bien, en la actualidad prácticamente cualquier hardware moderno tiene las capacidades de procesamiento paralelo en la forma de procesadores de múltiples núcleos, sin embargo, no toda aplicación se presta para utilizarlos de manera óptima, ya que las consideraciones de diseño del código al hacer uso de hilos y múltiples procesos en una misma aplicación son muchas y complejas a la vez. Se tiene que considerar el estado de la aplicación, la comunicación entre hilos y procesos, así como la seguridad de la memoria en la que se está trabajando, y en el caso de algunas aplicaciones es prácticamente imposible hacer uso de estos recursos adicionales debido a estas implicaciones en el desarrollo de un programa.

El raytracing por otro lado, es el perfecto ejemplo de una aplicación a la que se puede aplicar el uso de procesamiento paralelo sin ningún tipo de complicación, simplemente asegurándonos de ciertos lineamientos en el desarrollo del programa. Una vez cargada la escena en la aplicación el estado de los datos almacenados en memoria es inmutable, es decir, no se hace ninguna modificación a las características de los objetos, su posición o material, y aunque en algunos casos como animaciones pueden llegar a cambiar, estos se hacen en un proceso controlado donde no se está procesando ninguna imagen. Por lo tanto, se puede decir que en un raytracer, se conoce el estado de la aplicación en todo

momento, a excepción de la información de los pixeles de la imagen que se está generando en ese momento. También se puede observar que la función **render_pixel** trabaja con una "copia" de la escena a la vez y el resultado de la función (el color del pixel que se generó) no depende en ningún momento del resultado de cualquier otra llamada a esta función. Es aquí donde se puede hacer gran uso del procesamiento paralelo, ya que el resultado almacenado jamás comparte memoria ni necesita intercambiar información con ninguna otra llamada a la función, efectivamente aislando el procesamiento de la información de cualquier otro proceso o uso de memoria compartida. De lo anterior, se puede deducir que la función **render_pixel** es la candidata ideal para hacer uso de procesamiento paralelo en nuestra aplicación. Hay por lo menos dos maneras de optimizar el programa para reducir el tiempo de espera del resultado, en este caso la imagen generada, utilizando técnicas de procesamiento paralelo. El primero y más sencillo de implementar es la utilización de los núcleos adicionales con los que cuenta el procesador con el que estemos trabajando, estos pueden ir desde 2 núcleos y hasta 16 en un equipo convencional, gracias a ciertas tecnologías en los procesadores modernos, hay hardware que puede procesar dos hilos simultáneos por núcleo y podemos hacer uso de este tipo de tecnología, así mismo, existe hardware no convencional con una mayor cantidad de núcleos, sin embargo, no se necesitará modificar la implementación para poder trabajar con cualquier tipo de hardware que se presente pues trabajaremos con los hilos (threads) nativos del sistema operativo que se utilice.

El segundo método es hacer uso de la computación distribuida, y es ampliamente utilizado en la industria, sin embargo, requiere de una implementación mucho más compleja y de conocimiento de la arquitectura cliente – servidor o bien de herramientas de virtualización y clusters. En este caso, se utilizan múltiples máquinas conectadas a una red para hacer uso de todo el hardware disponible, este enfoque tiene la ventaja de poder escalar casi ilimitadamente dependiendo de las necesidades o del hardware en cuestión. En el caso de la implementación es el mismo, la función **render_pixel** es la candidata para realizar la distribución de la carga de procesamiento, y en este caso lo que se requiere es separar nuestra implementación en dos partes, la primera es un servidor maestro que cargue la escena y almacene el resultado final, la segunda parte es

el cliente o servidor esclavo, el cual se conecta al servidor maestro y solicita trabajo que realizar, así como la información precargada a utilizar, el servidor maestro a su vez le indica en que coordenadas o rango de coordenadas de la imagen tendrá que trazar los pixeles y devolver el resultado final una vez concluido el trabajo, el nodo principal es el encargado de distribuir el trabajo uniformemente según la cantidad de trabajadores que tenga para optimizar los recursos de hardware. Como se puede apreciar en la figura 4.3.1, este esquema requiere de más componentes y distintas implementaciones para poder funcionar como se plantea, pero las ventajas que ofrece son significativas cuando la carga de procesamiento es masiva, como en el caso de animaciones o películas donde unos cuantos segundos de video se forman a partir de cientos de imágenes que se generan con este tipo de técnicas.

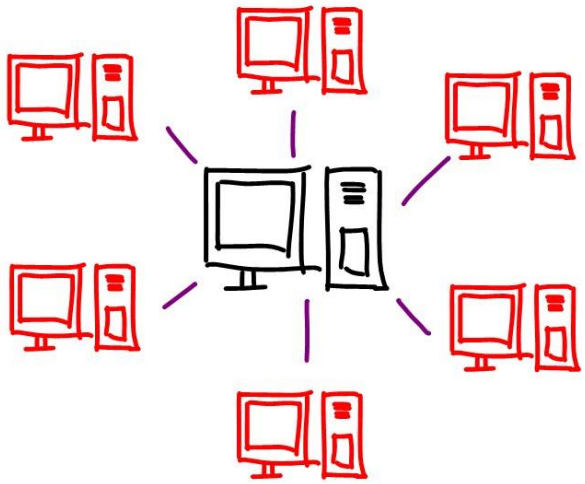


Figura 4.3.1 Cómputo distribuido

En el caso de la implementación que se realizó con las prácticas propuestas, se plantea cómo optimizar utilizando el primer caso, donde se utilizan hilos nativos del sistema operativo y procesamiento paralelo para realizar la misma cantidad de trabajo en magnitudes de tiempo menor dependiendo del hardware en el que ejecutemos la aplicación. Es importante notar que los resultados e implementación son muy distintos dependiendo de la plataforma y lenguaje de programación que se utilice.

La implementación del procesamiento paralelo depende de tres factores primarios, el primero es el hardware que se va a utilizar, tendremos que asegurarnos que nuestro procesador tiene múltiples núcleos o bien puede procesar varios hilos simultáneamente, de lo contrario, la utilización de hilos extra sin la capacidad de procesamiento adecuada podría incluso degradar el rendimiento de la aplicación. El segundo factor es el sistema operativo, y aunque todos los sistemas operativos modernos soportan hilos de procesamiento en paralelo, es importante hacer notar que se deben de investigar las implicaciones de usarlos en cada caso. El tercero y más importante es el lenguaje de programación a utilizar, no todos los lenguajes de programación cuentan con las capacidades de procesamiento paralelo, en especial los lenguajes interpretados (scripting), pues el intérprete se ejecuta en un solo proceso y mantiene un estado global del programa que se está ejecutando. Es muy importante haber escogido un lenguaje adecuado en nuestra implementación si se quiere sacar provecho de las capacidades del hardware, los lenguajes compilados como C son idóneos para la programación de un raytracer, pues no sólo cuentan con las librerías y acceso al sistema que nos permitirán usar al máximo los núcleos del procesador, sino también son magnitudes de tiempo más rápidos que otros lenguajes en la mayoría de los cálculos que se hacen dentro del algoritmo pues el programa está compilado nativamente en cada plataforma.

Como se mencionó en la introducción, la función **render_pixel** trabaja de manera independiente y es la que se utilizará como punto de partida para paralelizar el procesamiento de la aplicación. Es importante entender que la creación y destrucción de hilos de procesamiento también tiene un costo computacional, por lo que podemos crear una función adicional **render_line** que trace una línea completa de la imagen, de este modo aprovecharemos más tiempo el hilo de proceso recién creado, para esto hay que modificar la función **render** y sólo iterar sobre el alto de la imagen, la función **render_line** iterará sobre el ancho de la imagen haciendo llamadas a **render_pixel** cada vez que sea necesario.

```
render_line(y)
    x desde 0 hasta ancho
        image[y*ancho+x] = render_pixel(x, y)
```

Es importante que el estado global de la aplicación esté disponible desde estas funciones o métodos de nuestra clase, en este caso, **render_line** almacenará el resultado directamente en la imagen, mientras que **render** sólo irá esperando a que regresen las llamadas a la función **render_line** para llamarla nuevamente.

render()

y desde **0** hasta **alto**

render_line(y)

Con esta sencilla modificación estaríamos listos para hacer uso del procesamiento paralelo, un último ajuste adicional sería primero que nada investigar la cantidad de hilos paralelos que puede manejar el procesador en el que se está ejecutando el programa, principalmente para saber cuántos hilos es necesario que cree el ciclo principal de la aplicación. Hay maneras de obtener este dato directamente del sistema operativo y dependiendo del lenguaje de programación que se utilice y las librerías que se requieran utilizar es relativamente sencillo obtenerlo directamente desde el código, de lo contrario se recomienda que se pueda especificar desde el archivo de descripción de escena con la palabra **threads** con un valor predeterminado de 2, debido a que es el menor número de núcleos en los procesadores convencionales en la actualidad. Con este dato y las modificaciones a los métodos **render_pixel** y **render_line**, lo único que queda es que el método o la función **render** ejecute **render_line** en un hilo de procesamiento separado y lo haga tantas veces como **threads** haya disponibles, después espere a que alguno termine su ejecución y vuelva a llamar **render_line** de manera paralela. De este modo el código se ejecutará de manera simultánea reduciendo considerablemente el tiempo necesario para el resultado. Se puede encontrar una implementación de este raytracer en lenguaje Go que implementa procesamiento paralelo en:

https://github.com/phrozen/rayito

Bibliografía:

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

4.4 Estructuras de datos

Introducción

En ciencias de la computación, una estructura de datos es una forma particular de organizar datos en una computadora para que pueda ser utilizado de manera eficiente existen diferentes tipos de estructuras de datos son adecuados para diferentes tipos de aplicaciones, y algunos son altamente especializados para tareas específicas.

Las estructuras de datos son un medio para manejar grandes cantidades de datos de manera eficiente para usos tales como grandes bases de datos y servicios de indización de Internet. Por lo general, las estructuras de datos eficientes son clave para diseñar algoritmos eficientes. Algunos métodos formales de diseño y lenguajes de programación destacan las estructuras de datos, en lugar de los algoritmos, como el factor clave de organización en el diseño de software.

Octree. Toda la geometría de la escena dentro del cubo. El cubo, denominado padre o nodo raíz, se puede dividir en ocho más pequeños, en lo sucesivo, descendientes, subnodos u octanos, como se muestra en la ilustración (Figura 4.4.1). A su vez, cada uno de los descendientes puede ser dividido en ocho consecutivos y así continuar recursivamente hasta alcanzar un determinado umbral, que es más a menudo la cantidad del número mínimo de polígonos que el cubo puede contener, o el nivel máximo de dividir la etapa, o la longitud mínima de los bordes del cubo. Cada descendiente ha asignado una lista de triángulos, que en sí contiene. La raíz contiene todos los polígonos del mundo. Cada uno de los padres contiene los polígonos, que posteriormente se asignan a los descendientes concretos.

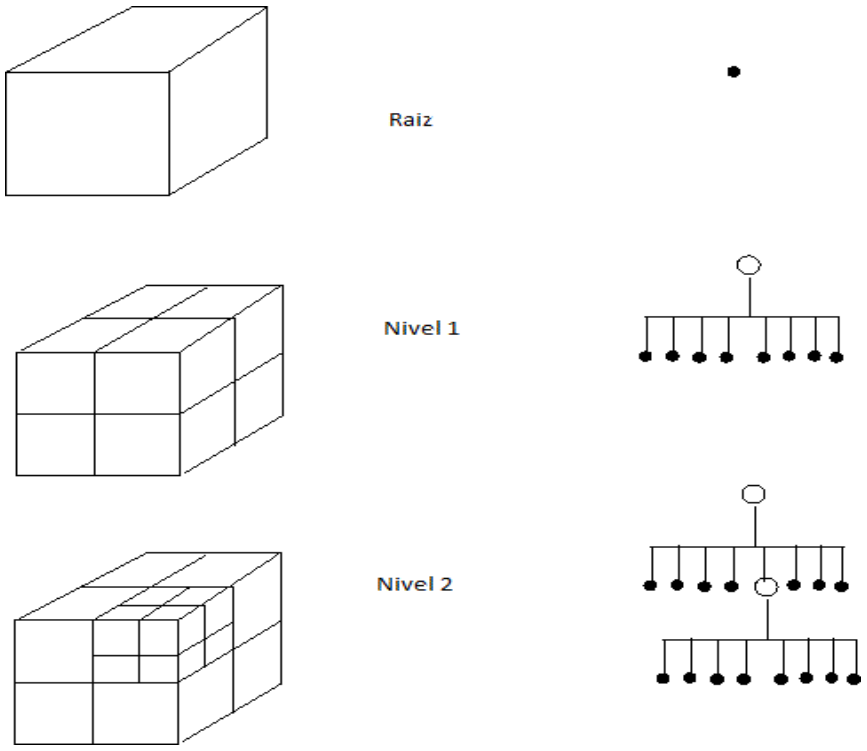


Figura 4.4.1 Octree de dos niveles[55]

Dibujar la escena depende de comprobar si un cubo que rodea la raíz está en el área de la vista. Si no lo es, entonces no se dibuja ningún polígono de la escena. Si es visible como un todo, entonces todos los polígonos de la escena se dibujan, por lo tanto, el mundo entero. Si la raíz es visible sólo parcialmente, comenzamos a recorrer el árbol. Comprobamos a los descendientes como lo hicimos con la raíz. Si nos encontramos con un cubo totalmente visible, rendimos la geometría contenida dentro de él. Si el cubo es visible sólo parcialmente y no tiene descendientes (subnodos), dibujamos polígonos asignados a él.

Al principio, debe crear un cubo que rodee la raíz de toda la escena. Para lograr esto, estamos buscando el mayor valor absoluto (lo llamamos V) para las coordenadas x, y, z de cada nodo de la lista de geometría de escena. Este valor servirá para generar un cubo

[55] http://www.forceflow.be/2012/04/20/ray-octree-traversal/

(los vértices opuestos son [V, V, V] y [-V, -V, -V]). Adjuntamos la lista de todos los polígonos de la escena que vamos a dividir a la estructura que describe la raíz. Entonces el cubo se divide en ocho secciones a lo largo del eje principal, y comprueba qué polígonos están en octanos diferentes (subnodos). Se procede así recursivamente hasta que alcance un cierto umbral en donde nos fijamos la condición de la terminación del árbol. Dado que un polígono puede estar contenido en unos pocos nodos, a través de unos cuantos cubos, está sumamente indicado que no se almacenaría una copia en unas pocas hijos. En algunos casos, esto forzará la tarjeta a varias veces el procesamiento del mismo polígono, y por lo tanto disminuirá la productividad. El contador de marcos en cada polígono será la solución. Rendering comprueba el valor del contador y previene un dibujo adicional del triángulo ya representado en la pantalla en el marco actual.

Kd tree

En un nivel alto, un árbol kd es una generalización de un árbol binario de búsqueda que almacena puntos en el espacio k-dimensional. Es decir, se podría usar un árbol kd para almacenar una colección de puntos en el plano cartesiano, en un espacio tridimensional, etc. También se podría usar un árbol kd para almacenar datos biométricos, por ejemplo, representando los datos como un Ordenada tupla, tal vez (altura, peso, presión arterial, colesterol). Sin embargo, no se puede usar un kd-tree para almacenar colecciones de otros tipos de datos, como cadenas. También tenga en cuenta que aunque es posible construir un árbol kd para contener datos de cualquier dimensión, todos los datos almacenados en un árbol kd deben tener la misma dimensión. Es decir, no se pueden almacenar puntos en el espacio bidimensional en el mismo árbol kd que los puntos en el espacio de cuatro dimensiones.

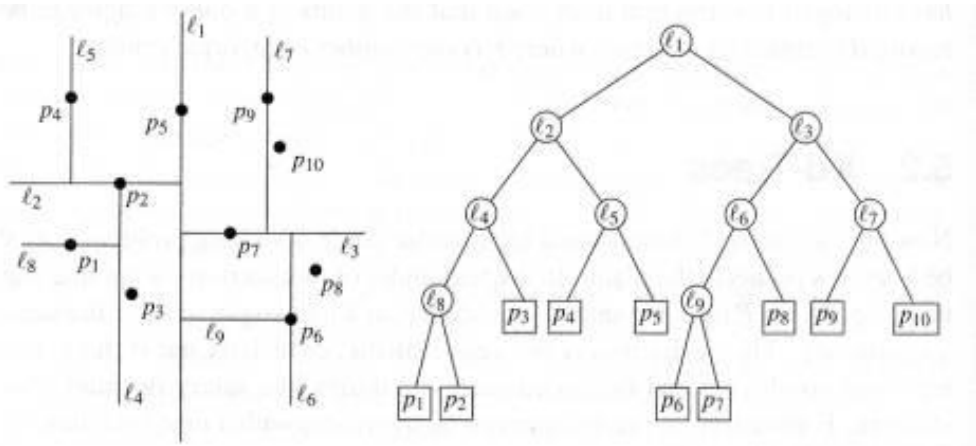


Figura 4.4.1 Kd tree ejemplo[56]

Un árbol k-d (abreviatura de árbol k-dimensional) es una estructura de datos de división de espacio para organizar puntos en un espacio k-dimensional. K-d árboles son una estructura de datos útiles para varias aplicaciones, tales como búsquedas que implican una clave de búsqueda multidimensional (por ejemplo, las búsquedas de rango y las búsquedas de vecinos más cercanos). K-d árboles son un caso especial de espacio binario partición árboles.

K-d árboles no son adecuados, sin embargo, para encontrar eficientemente el vecino más cercano en espacios de alta dimensión. Como regla general, si la dimensionalidad es k, el número de puntos en los datos, N, debe ser N »2k.

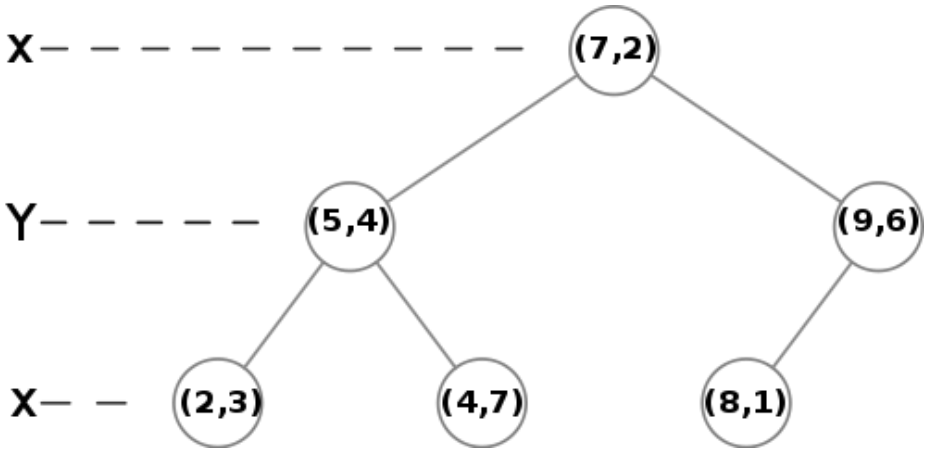
De lo contrario, cuando se usan árboles k-d con datos de alta dimensión, se evaluará la mayor parte de los puntos del árbol y la eficiencia no será mejor que la búsqueda exhaustiva, y en su lugar se utilizarán otros métodos, como el vecino más cercano aproximado.

[56] http://homes.ieu.edu.tr/hakcan/projects/kdtree/kdTree.html

El árbol se construye, para empezar, tiene un conjunto de puntos en un espacio k-dimensional. Demos un ejemplo de un árbol k-d de 2 dimensiones:

Entrada: (2,3), (5,4), (9,6), (4,7), (8,1), (7,2)

Salida: Un árbol bidimensional k-d [2]:



En el caso de árboles binarios de búsqueda, la partición binaria de la línea real en cada nodo interno está representada por un punto en la línea real. De manera similar, en el caso de un árbol k-d de 2 dimensiones, la división binaria del plano cartesiano bidimensional en cada nodo interno está representada por una línea en el plano.Por lo tanto, en el caso de árboles de búsqueda binaria, el punto representado por el nodo interno sirve como el punto utilizado para particionar la línea real. Para seleccionar una línea divisoria en el caso de árboles bidimensionales k-d, esencialmente, se podrá elegir cualquier línea que pasa a través del punto representado por el nodo interno para dividir el plano cartesiano bidimensional.

La salida del árbol k-d anterior se ha construido usando un método simple para elegir la línea de partición en cada nodo interno del árbol:

Nivel 0: - Elija la línea de partición perpendicular a la primera dimensión (X en este caso) y pasando por el punto representado por el nodo en cuestión.

Nivel 1: - Elija la línea de partición perpendicular a la segunda dimensión (Y en este caso) y pasando por el punto representado por el nodo en cuestión.

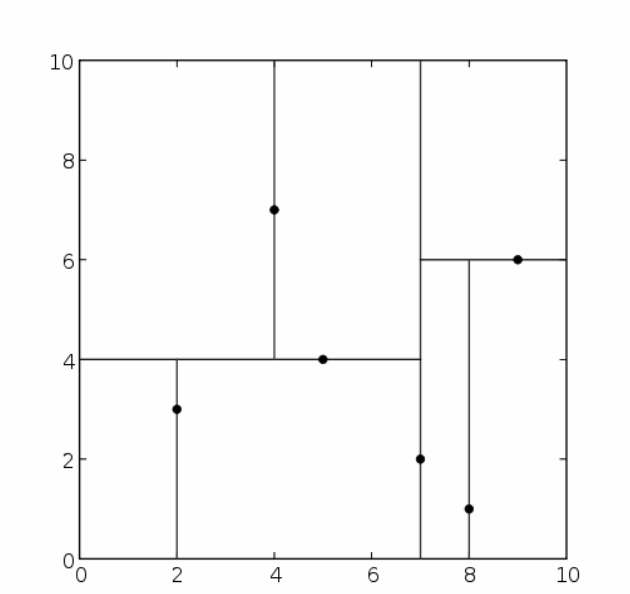
Nivel k-1: - Elija la línea de partición perpendicular a la k-ésima dimensión y pasando por el punto representado por el nodo en cuestión.

Nivel k: - Elija la línea de división perpendicular a la primera dimensión (X en este caso) y pasando por el punto representado por el nodo en cuestión.

Así que básicamente, en cada nivel alternamos entre las dimensiones X e Y para elegir una línea divisoria en cada nodo interno del árbol k-d.

Las etiquetas que ve al lado de cada uno de los nodos del árbol k-d representan la elección de la dimensión para la línea de partición en los nodos de ese nivel.

Veamos ahora cómo nuestro árbol k-d bidimensional divide el plano bidimension



Bibliografia

Peter Shirley, Michael Ashikhmin, Steve Marschner; Fundamentals of Computer Graphics; CRC Press, Jul 21, 2009

5 Conclusiones

- En el mundo de hoy con el vertiginoso avance de las tecnologías informáticas. Resulta de vital importancia el aprendizaje. Por lo cual tener una herramienta didáctica de apoyo a la docencia es fundamental para el proceso de aprendizaje de los alumnos a nivel licenciatura.
- El campo de las ciencias computacionales está en constante expansión por lo cual es indispensable tener más recursos de aprendizaje para complementar el conocimiento.
- Los conceptos computacionales son mejor aprendidos cuando se da la conjunción entre la teoría y la práctica. La unión permite fortalecer tanto la parte teórica como la práctica. La ingeniería es esencialmente una actividad en donde siempre existe una fuerte dependencia de ahí la importancia de favorecer esta interacción.
- Para aprender temas más avanzados de computación gráfica es indispensable tener las bases sólidas; es decir, los antecedentes. Ya que con buenas bases se puede ir avanzando a nuevos conceptos de una mejor forma.
- Es importante saber cómo funciona la realidad física y aprender los modelos que describen su funcionamiento. También adaptar los conceptos de un campo a otro que se rige con diferentes reglas. En nuestro caso imágenes digitales en una computadora.
- Los algoritmos son un tema fundamental para poder desarrollar un sistema computacional. Ya que estos permiten simplificar muchos procesos, además de aprovechar los recursos limitados de las computadoras.
- El seudocódigo es vital para proyectos de complejidad elevada debido a que este proporciona una forma estructurada de ir codificando.
- La importancia de las pruebas para hacer un código robusto y con la funcionalidad deseada.
- La relevancia de segmentar un proyecto complejo para así ir desarrollando un producto que cumpla con todos los requerimientos, ya que se minimizan los fallos y cuando suceden estos, son fáciles de solucionar.

- La importancia de diseñar estrategias de aprendizaje que permitan al alumno ir de lo básico a lo complejo bajo una secuencia lógica para integrar todo el trabajo y conocimiento.
- La relevancia de los lenguajes orientados objetos en los paradigmas actuales de la computación, ya que estos permiten crear sistemas complejos además permitir una representación del mundo real.
- La importancia de conocer el álgebra vectorial para poder realizar las operaciones que se requieren en el trazado de rayos del raytracer.
- Entender cómo se construye una imagen digital en la computadora para así poder crear las imágenes que construirá el raytracer debido a que se tienen los fundamentos.
- Entender el concepto de la escena virtual y la cámara (punto de vista) para poder crear imágenes con el algoritmo raytracer.
- Entender cómo funcionan las luces y las sombras dentro del algoritmo porque estos dos aspectos funcionan distintos en la realidad física.
- La importancia de los conceptos *shading* y la recursividad para crear imágenes más complejas con mayor textura que dan como resultado al ojo humano; imágenes más realistas.
- Dominar el concepto de reflexión y refracción además de su implementación para que el algoritmo sea capaz de crear objetos realistas. Muchos objetos tienen textura y transparencia. La luz se va a comportar de distinta forma dependiendo de cómo y contra que se impacta. Estos conceptos permiten replicar objetos del mundo físico.
- La relevancia del *oversampling* para crear acabados en las imágenes que resultan en productos con una mayor claridad.
- La importancia de desarrollar un proyecto que tiene una estructura y que puede ser fácilmente escalable, aprendiendo y agregando otros conceptos, ya que se tiene una base sólida.

6 BIBLÍOGRAFIA

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

Ray Tracing from the Ground Up – Kevin Suffern, 2007.

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995

CASTAÑEDA De I. P., Érik, Geometría analítica en el espacio México Facultad de Ingeniería – UNAM 2003.

FOLEY, James D,; DAM VAN, Andries; FEINER, Steven K,; HUGHES, John F, Computer Graphics: Principles and Practice in C (2nd Edition) 2a. Edición USA Portland Addison-Wesley Pub Co, 1995.

Eric Haines, Pat Hanrahan, Robert L. Cook, James Arvo, David Kirk, Paul S. Heckbert,

An Introduction to Ray Tracing (The Morgan Kaufmann Series in Computer Graphics) (1st Edition)

Academic Press; 1st edition (February 11, 1989)

Jhon F. Hughes, Computer Graphics Principles and Practice, tercera edición, 2014

Manual de iluminación, de Eli Sirlin, publicado por el INT, 2005 y Ed. Atuel, 2006;

Fuentes electrónicas

https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful
https://www.lanshor.com/interseccion-rayo-triangulo/
http://www.bdigital.unal.edu.co/9424/1/1077446730.2013.pdf
http://publicaciones.unisimonbolivar.edu.co/rdigital/inovacioning/index.php/identic/article/viewFile/20/28
http://cphoto.uji.es/siu020/ewExternalFiles/Tema_01_Modelado_Poligonal.pdf
https://www.quora.com/What-is-a-mesh-in-OpenGL

Apéndice 1 – Metodología

La selección de una metodología es muy importante en el desarrollo de cualquier proyecto de software, en especial si tienen un tamaño importante o bien debido a su complejidad. En el caso de la programación de un ***raytracer*** se pueden utilizar diversas metodologías para completarlo y aunque es recomendado que usuarios experimentados utilicen aquella con la que se sientan más cómodos, se recomiendan dos de ellas dependiendo de la experiencia que tenga el desarrollador.

La metodología recomendada para los programadores experimentados es *Test-Driven Development (TDD)*, esta metodología plantea escribir las pruebas de código necesarias antes de desarrollar la funcionalidad deseada. La finalidad es escribir la menor cantidad de código necesario para cumplir cada prueba escrita, sin embargo, puede darse el caso de que un código erróneo satisfaga alguna de las pruebas, en este caso hay que agregar parámetros adicionales y chequeos a las pruebas y reescribir el código. La metodología *TDD* es ampliamente utilizada en la industria hoy en día, y existen un sin número de librerías dedicadas a la realización de pruebas de unidad, coherencia (regresión) y desempeño, dependiendo de la plataforma y lenguaje que se utilice. La finalidad de la metodología es que la base de código se pueda corregir y aumentar paulatinamente sin romper en ningún momento la funcionalidad de esta, escribiendo cada vez código más eficiente para la realización de cada tarea.

El ***raytracer*** se escribe de manera modular, por esto se recomienda utilizar un lenguaje orientado objetos, aunque no es estrictamente necesario. Cada clase dentro del programa, encapsula la funcionalidad de una cosa en específico dentro de la aplicación, y es por esto que cada clase se puede escribir y probar de manera individual como se hace a lo largo de las prácticas. De este modo, el desarrollador puede escribir las pruebas de cada módulo por separado antes de implementar la funcionalidad de cada uno de ellos, aunque para esto muchas veces es necesario conocer cuál será el resultado final esperado de cada uno de los módulos y si no se está familiarizado con la teoría del ***raytracing*** esto se vuelve muy complejo y contraproducente.

Es por esto que aunque la metodología *TDD* es la recomendada si se tiene la experiencia necesaria en el tema, se vuelve una sobrecarga de trabajo y aprendizaje que se salen del alcance del proyecto. Además, dependiendo de la práctica y del lenguaje que se haya utilizado para el desarrollo, se requieren de herramientas y librerías adicionales para realizar este tipo de pruebas previas, lo cual incrementa la complejidad del proyecto en su totalidad. La metodología *TDD* puede estar mejor ubicada en ciertos proyectos donde se conoce la teoría y la funcionalidad que se quiere lograr previo a comenzar el proyecto de software, así se tiene una visión clara del

objetivo desde un principio y se puede asegurar de antemano que el código escrito es el mínimo necesario y funciona correctamente. Sin embargo, en el caso del algoritmo del ***raytracing***, la complejidad del proyecto se encuentra concentrada en la parte teórica del algoritmo y en las matemáticas necesarias para los cálculos de iluminación y colisiones, así como en los conceptos de computación gráfica como la generación de imágenes y las operaciones con vectores y colores. De este modo cualquiera que no esté familiarizado previamente con estos conceptos no tendrá previamente definido el objetivo al que se quiere llegar en cada práctica y cada módulo, haciendo que la creación de pruebas al respecto se vuelva tediosa y complicada.

Por esto, se recomienda utilizar una metodología más convencional para todos aquellos que no estén familiarizados de antemano con la teoría del algoritmo. El ***Incremental build model*** es una metodología que se adapta perfectamente a las necesidades del proyecto. Se basa en los elementos del método en cascada junto con la filosofía iterativa del prototipado de aplicaciones. El concepto es muy simple, el proyecto se diseña, se implementa y se prueba agregando un poco más de funcionalidad en cada iteración de manera incremental, es fácil identificar los errores que se puedan haber introducido a la aplicación pues se incrementa una pequeña cantidad de código en cada iteración, asegurándonos de no continuar con la siguiente implementación hasta que la base de código funciona correctamente. Si se presta atención al desarrollo de las prácticas, estas se encuentran diseñadas de esta manera, pues si bien, para la utilización de esta metodología es muy recomendado tener cuidado en realizar un buen diseño de todo el proyecto de antemano, en el caso del ***raytracer*** y del proyecto que se plantea, ya está hecho de esta manera. Gracias a esto, cualquier desarrollador puede realizar las prácticas una a la vez con la confianza de que el proyecto completo está diseñado correctamente en su totalidad, y puede enfocarse en la implementación de cada uno de los módulos, probando posteriormente los resultados y asegurándose de que la implementación fue la adecuada antes de proceder a la siguiente práctica.

Debido a que tanto el ***raytracer*** como las prácticas están escritas de manera modular y se avanza a través de los conceptos de manera paulatina, el modelo de construcción incremental *(Incremental Build Model)* se adapta a la perfección. A lo largo del desarrollo de la aplicación el lector se puede familiarizar con los conceptos complejos durante cada una de las prácticas e implementarlos de manera progresiva sin preocuparse por el diseño del proyecto en su totalidad y probar la correcta implementación de cada módulo antes de continuar con el siguiente. Entre las ventajas que provee esta metodología, es el poder detenerse en cualquier parte del proyecto el tiempo necesario para asegurarse de tener los conceptos claros y una implementación sólida,

ya que el proyecto está diseñado en su totalidad con antelación, no es necesario conocer de antemano los módulos subsecuentes y conceptos teóricos avanzados para poder comenzar a diferencia de otras metodologías como la *TDD*.

Apéndice 2 – Rendimiento

El algoritmo del ***raytracing*** es por su naturaleza iterativa y recursiva muy demandante en recursos, principalmente procesamiento, debido a esto es importante no solo escribir código eficiente, sino también escoger una plataforma adecuada que aproveche al máximo los recursos a su disposición. Las prácticas son agnósticas, quiere decir que no asumen ningún lenguaje de programación específico o plataforma, sin embargo, recomiendan la utilización de un lenguaje orientado a objetos por la facilidad de comprensión e implementación de los conceptos que explican. De este modo, la implementación de referencia del ***raytracer*** está escrita en Ruby[57], se escogió Ruby como lenguaje de programación, no por su desempeño, pues es un lenguaje *scripting*, y como tal tiene muchas deficiencias en esa área, sino por su facilidad de comprensión del código pues es un lenguaje muy humano, quiere decir, es sencillo entender el algoritmo al revisar la implementación.

De todos modos, en este apéndice se realizarán algunas pruebas de desempeño con la implementación de referencia en Ruby, así como con una implementación del proyecto hecha en Go[58], un lenguaje compilado de propósito general con el cual nos es sencillo agregar la funcionalidad de procesamiento paralelo y aprovechar al máximo los recursos de procesamiento disponibles. Cabe mencionar, que aunque eficiente, el código de ambas implementaciones no está optimizado al máximo, pues al hacerlo, complica mucho el código de referencia que se presenta, lo cual iría en contra del propósito del proyecto de ser un ejercicio didáctico para cualquiera con nociones básicas de programación. El objetivo final es mostrar y explicar detalladamente el funcionamiento del algoritmo del ***raytracing***, así como los conceptos de computación gráfica que están inherentemente relacionados con él. Aun así, estas pruebas de desempeño nos dan un panorama más amplio acerca de la importancia de la selección de la plataforma y del lenguaje de programación para un proyecto más complejo. Ambas implementaciones están disponibles en internet para su utilización y revisión.

Para las pruebas se utilizará el mismo equipo y plataforma de referencia con el fin de poder equiparar ambas implementaciones y replicarlas posteriormente. El equipo de referencia cuenta con un procesador Intel Core i7 3770K @ 3.5 Ghz de reloj base, es necesario hacer la aclaración pues la mayoría de los procesadores modernos tienen una modalidad *Turbo* donde incrementan

[57] https://www.ruby-lang.org/en/
[58] https://golang.org/

la velocidad del reloj para ciertas tareas que utilizan uno solo núcleo. Este modelo de procesador cuenta con 4 núcleos físicos y la tecnología *Hyper Threading* de Intel, la cual permite ejecutar 2 hilos de procesamiento a la vez, con lo cual podemos evaluarlo como un equipo con 8 núcleos lógicos para fines prácticos. También cuenta con un disco sólido Corsair GT y 16 GB de RAM Corsair Vengeance DDR3 @ 1866 Mhz, aunque la velocidad del disco y de la memoria no son el principal problema en cuanto al algoritmo se refiere, es necesario tomar nota para poder replicar estas pruebas posteriormente y hacer una comparación más acertada contra otros equipos. Por último, se utiliza Windows 10 en su versión Pro de 64 bits.

La versión de Ruby que se utiliza es la última al momento de hacer las pruebas.

```
ruby 2.3.3p222 (2016-11-21 revision 56859) [x64-mingw32]
```

La versión de Go que se utiliza es la última al momento de hacer las pruebas.

```
go version go1.8 windows/amd64
```

La escena que se utilizará como referencia para las pruebas es con la que se trabajó al final del proyecto 2 y se incluye debajo.

```
image_size 1920 1200

depth 5

oversampling 2

field_of_view 45

camera_position 5.0 5.0 5.0
camera_look 1.0 1.0 0.0
camera_up 0.0 0.0 1.0

sphere 3 3.0 1.5 0.8 0.8
sphere 4 0.0 0.0 1.0 1.0
sphere 2 1.0 3.0 0.7 0.7
sphere 0 -1.0 1.5 0.5 0.5

plane 1 0.0 0.0 1.0  0.0

light point -2.0 1.0 4.0 1.0 1.0 1.0
light point 2.0 13.0 4.0 0.5 0.5 0.5
light ambient 0.0 0.0 0.0 0.1 0.1 0.1

material 1.0 0.4 0.0 0.9 0.8 2.0 0.15 0.0 0.0
material 0.8 0.8 0.9 1.0 0.0 4.0 0.1 0.0 0.0
material 1.0 0.2 0.2 0.9 0.7 1.0 0.2 0.0 0.0
material 0.2 1.0 0.2 0.4 0.8 2.0 0.15 0.0 1.1
material 0.3 0.3 1.0 0.7 0.4 0.8 0.3 0.0 0.0
```

Para que las pruebas sean significativas, se utilizarán parámetros altos para que el programa se ejecute más tiempo y poder obtener resultados más estables, unas pruebas de unos pocos segundos tienen mucha varianza debido a los demás procesos que se ejecutan en el equipo y el sistema operativo. Se utilizará una resolución de 1920x1200, así como profundidad de trazado de 5 y oversampling de 2x2. Por último, aprovecharemos el comando *Measure-Command* integrado en *PowerShell* de Windows para medir precisamente el tiempo de ejecución de cada versión.

Primero ejecutaremos la versión de Ruby de referencia, la cual se debe de tener al término del proyecto 2 y puede encontrarse en:

https://git.io/vDUAu

```
Measure-Command {ruby .\proyecto2.rb .\prueba.txt}

Days              : 0
Hours             : 0
Minutes           : 9
Seconds           : 43
Milliseconds      : 724
Ticks             : 5837241874
TotalDays         : 0.0067560669837963
TotalHours        : 0.162145607611111
TotalMinutes      : 9.72873645666667
TotalSeconds      : 583.7241874
TotalMilliseconds : 583724.1874
```

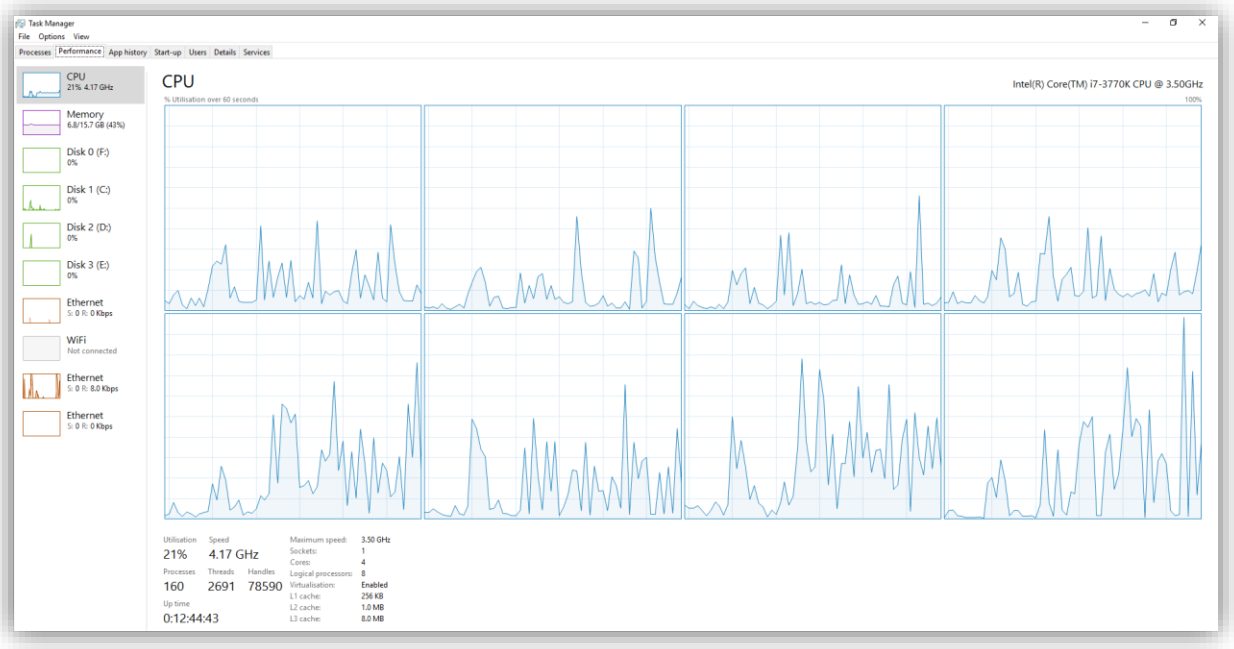


Figura A. Ejecución raytracer en Ruby

El tiempo total de ejecución es de 9 minutos con 43 segundos. Arriba se muestra una captura de pantalla del administrador de tareas donde se puede observar que la utilización del CPU estuvo en el 20% en promedio (18-22%) durante la ejecución, también se puede notar que el CPU utilizó su función *Turbo* durante la ejecución con una velocidad máxima de 4.17 Ghz, pues no se utilizaron los núcleos al máximo.

La siguiente prueba es de la versión del proyecto 2 programada en en el lenguaje Go y compilada para la plataforma Windows x64, se ejecutará con la restricción de utilizar un solo hilo de procesamiento para equiparar ambas implementaciones. La versión en Go puede encontrarse en:

https://github.com/phrozen/rayito

```
Measure-Command {./rayito -workers=1 -file="samples/prueba.txt"}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 7
Milliseconds      : 995
Ticks             : 79950589
```

```
TotalDays         : 9.25354039351852E-05
TotalHours        : 0.00222084969444444
TotalMinutes      : 0.133250981666667
TotalSeconds      : 7.9950589
TotalMilliseconds : 7995.0589
```

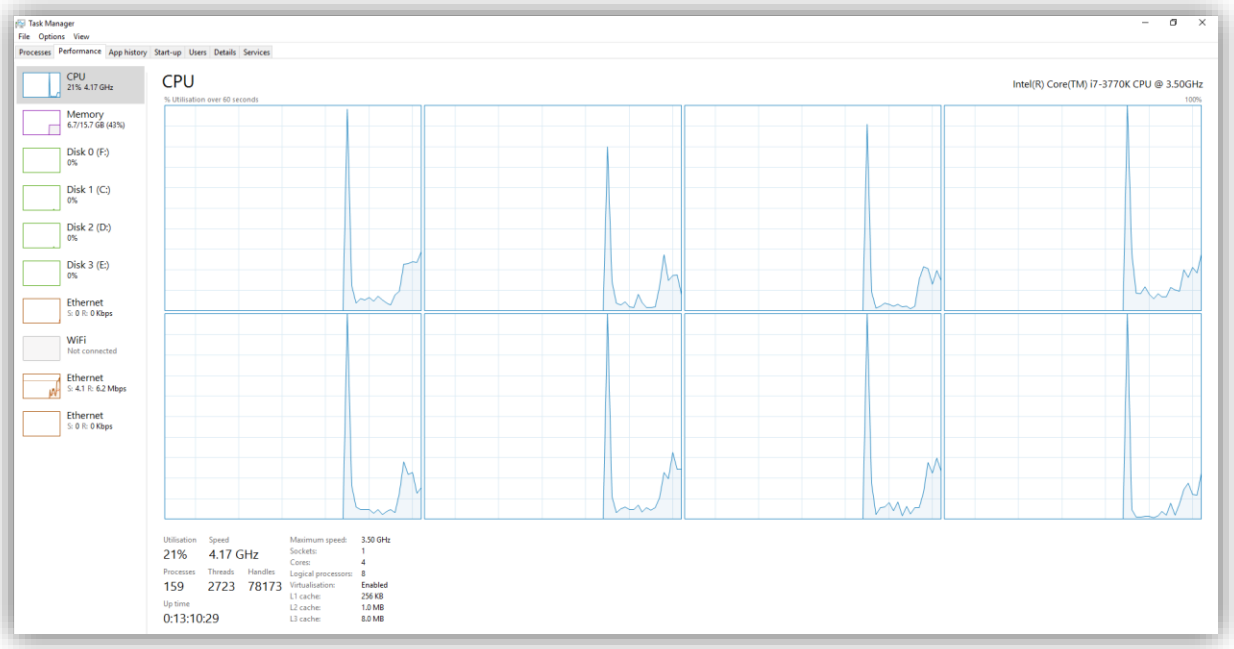


Figura B. Ejecución raytracer en Go, 1 hilo

Se puede observar de la imagen anterior que la versión en Go utiliza los mismos recursos que la versión en Ruby, aproximadamente el 20% del CPU y la velocidad de este se incrementa a 4.17 Ghz con el fin de aprovecharlo al máximo. Sin embargo, la versión de Go se ejecuta en tan solo 7 segundos. Esta enorme diferencia no es un error, y se debe principalmente al lenguaje. Ruby es un lenguaje *scripting* que se ejecuta a través de un intérprete el cual dinámicamente va solicitando la memoria conforme realiza las operaciones necesarias, este tipo de aplicaciones con una gran cantidad cálculos se ejecutan pobremente en este tipo de lenguajes interpretados, pues pierden más tiempo haciendo movimientos en la memoria que ejecutando realmente los cálculos necesarios. Por otro lado, la versión en Go está compilada nativamente a la plataforma y aprovecha al 100% los recursos que le permite el sistema operativo, haciendo que la ejecución sea más rápida hasta en un factor de 100.

Por último, se ejecutará la versión de Go utilizando todos los núcleos de procesamiento disponibles, sin embargo, esta prueba toma alrededor de 2 segundos en completarse y no se puede apreciar la utilización del CPU al 100%, por lo cual, se modificarán algunos parámetros de

la escena para exigirle trabajo al equipo. Simplemente se duplican los valores de resolución, profundidad de trazado y oversampling. Esta simple modificación realizará más de 16 veces más cálculos pues se incrementa de manera exponencial.

```
image_size 3840 2400
depth 10
oversampling 4
```

Se ejecuta nuevamente el comando:

```
Measure-Command {./rayito -workers=8 -file="samples/prueba.txt"}

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 28
Milliseconds      : 176
Ticks             : 281765478
TotalDays         : 0.000326117451388889
TotalHours        : 0.00782681883333333
TotalMinutes      : 0.46960913
TotalSeconds      : 28.1765478
TotalMilliseconds : 28176.5478
```

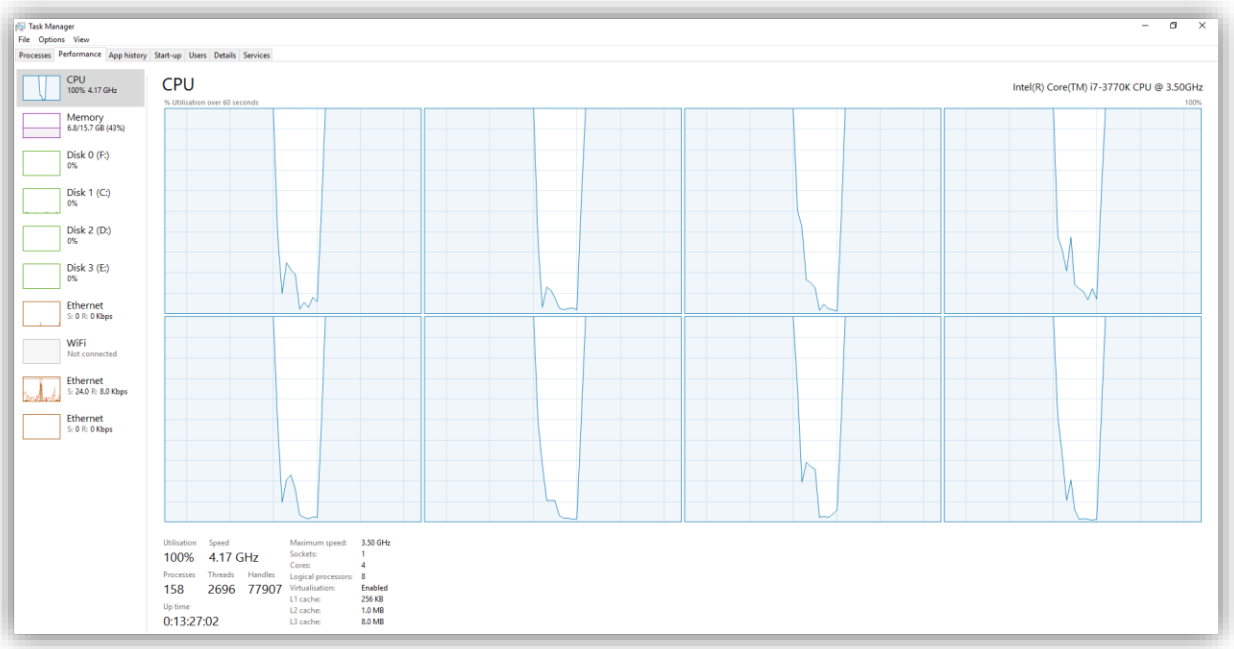


Figura C. Ejecución raytracer Go 8 hilos

De los datos anteriores se puede observar que la ejecución tarda tan sólo 28 segundos y en la imagen del administrador de tareas el CPU se utiliza al 100% pues se utilizan los 8 núcleos disponibles. Go hace trivial la tarea de agregar *multithreading* a las aplicaciones y se utiliza principalmente en la industria para aplicaciones web de alto desempeño, principalmente servidores. La imagen final generada por ambas versiones es idéntica y se muestra debajo.

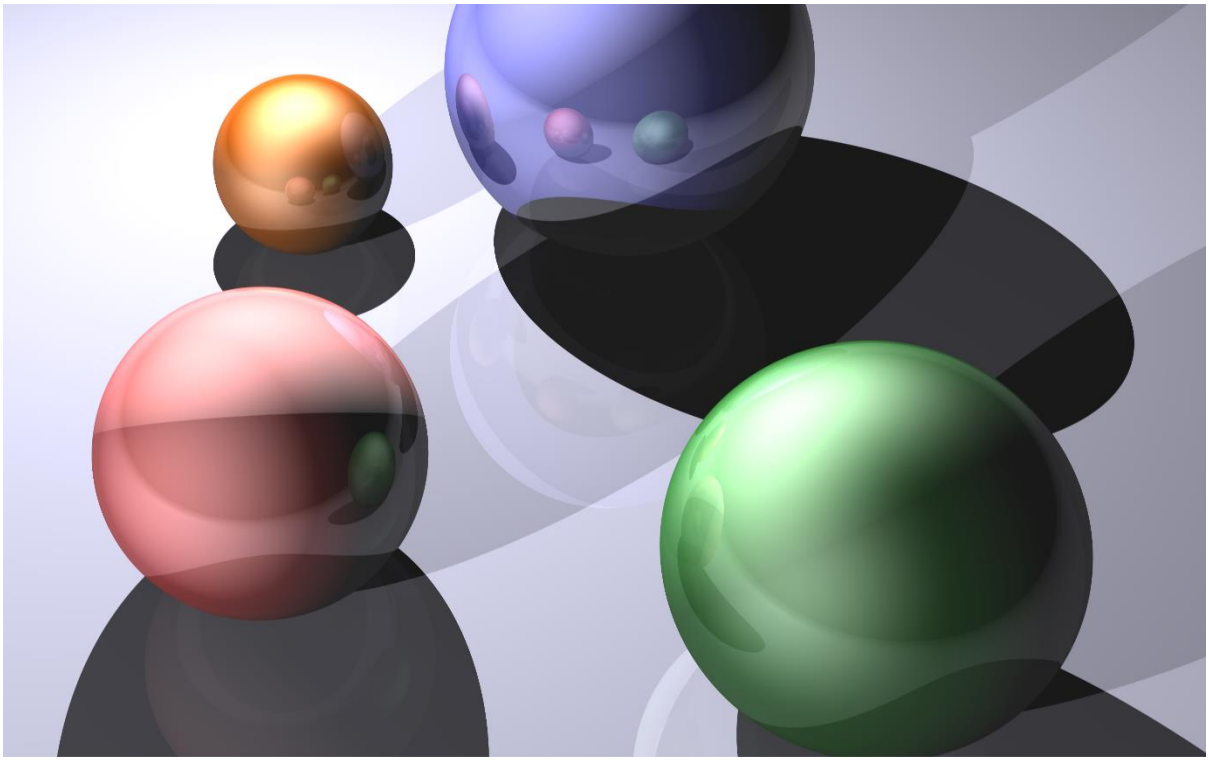


Figura D. Imagen generada por el raytracer

En conclusión, con estas pruebas se puede constatar la importancia de la selección del lenguaje de programación, no sólo para el ***raytracing*** sino para cualquier aplicación o desarrollo que implique una cantidad masiva de procesamiento. Go es un lenguaje compilado nativamente, así como C y C++ que siguen siendo los lenguajes ideales para aplicaciones de alto rendimiento en computación gráfica, y aunque el desempeño de Go se encuentra ligeramente por debajo de estos, la agilidad de programación en el lenguaje y su flexibilidad lo hacen un candidato ideal para programar aplicaciones como servidores o procesamiento digital.

ANEXO 1

```
# Proyecto 2
# Ray Tracer Avanzado
# Guillermo Estrada
# Rene Garcia
# Edgar López
# Luis Ledezma
# Daniel Castillo
#
# Constantes utilizadas dentro del programa
MAX_DISTANCE = 9999999999
PI_OVER_180 = 0.017453292
SMALL = 0.000000001
# Práctica 1
# Definimos nuestra clase Vector3 y sus métodos
class Vector3
  # Se definen x, y, z como de solo lectura publicamente en la clase
  attr_accessor :x, :y, :z
  # Constructor de la clase, se inicializa Vector3.new(x, y, z)
  # los valores predeterminados son 0.0 -> Vector3.new()
  def initialize(x = 0.0, y = 0.0, z = 0.0)
    @x, @y, @z = x, y, z
  end
  # Producto punto
  def dot(v)
    return @x*v.x + @y*v.y + @z*v.z #escalar
  end
  # Producto cruz
  def cross(v)
    r = Vector3.new()
    r.x = v.y*@z - v.z*@y
    r.y = v.z*@x - v.x*@z
    r.z = v.x*@y - v.y*@z
```

```
  return r
end
# Módulo de un vector
def module()
  # La raiz cuadrada 'sqrt' esta en la libreria 'Math'
  return Math.sqrt(@x*@x + @y*@y + @z*@z)
end
# Normalizar el vector
# En Ruby es común que los metodos que modifican el objeto
# terminen con ! en su nombre
def normalize!()
  m = self.module()
  # Dividimos cada componente del vector entre el modulo
  # si este es diferente de 0. (x /= m) --> (@x = @x/m)
  if m != 0.0
    @x /= m; @y /= m; @z /= m;
  end
  return self
end
# Operator overloading, para poder sumar, restar o multiplicar
# vectores utilizando +, -, *
# Suma de Vectores
def +(v)
  return Vector3.new(@x+v.x, @y+v.y, @z+v.z)
end
# Resta de vectores
def -(v)
  return Vector3.new(@x-v.x, @y-v.y, @z-v.z)
end
# Multiplicación por escalar
def *(s)
  return Vector3.new(@x*s, @y*s, @z*s)
end
# to_s opcional para debugging imprime vector en formato texto
def to_s
```

```
    return "[#{@x}, #{@y}, #{@z}]"
  end
end
# Práctica 2
# Definimos nuestra clase Color que incluye las componentes RGB
# que forman un pixel. En nuestro caso de uso, las componentes
# estan en el rango [0.0, 1.0] para poder hacer operaciones compatibles
# con vectores. Para generar la imagen deberemos transformarlas
# al rango [0, 255] que es 8 bits por componente.
class Color
  # Definimos acceso publico de solo lectura para las componentes
  # RGB que son los canales de color Red Green Blue
  attr_accessor :r, :g, :b

  # Constructor de la clase, se inicializa Color.new(r, g, b)
  # los valores predeterminados son 0.0 -> Color.new() color negro
  def initialize(r = 0.0, g = 0.0, b = 0.0)
    @r, @g, @b = r, g, b
  end
  # Operator overloading, para poder sumar o multiplicar
  # colores utilizando +, *
  # Suma de colores, se suman cada componente por separado
  # devolvemos un nuevo color sin modificar el original
  def +(c)
    return Color.new(@r + c.r, @g + c.g, @b + c.b)
  end
  # Multiplicación de color por escalar (factor), se multiplica cada
  # componente por el valor del escalar y devolvemos un nuevo color.
  def *(f)
    return Color.new(@r *f, @g * f, @b * f)
  end
  # Sobre escribimos el método to_s (to string) de Ruby, que se usa
  # para especificar como convertir el objeto a texto. Aquí es donde
  # convertiremos cada color del rango [0.0, 1.0] al rango [0, 255]
  # y lo devolveremos en cadena de texto para su escritura en el
```

```
  # formato PPM.
  def to_s()
    # Multiplicaremos cada componente por 255 y acotaremos el valor final
    # para que se este dentro del rango, sólo regresaremos la parte entera
    # del valor final (%d) en cadena de texto.
    return "%d %d %d" % [ [[255.0, @r*255.0].min, 0.0].max.to_i,
                          [[255.0, @g*255.0].min, 0.0].max.to_i,
                          [[255.0, @b*255.0].min, 0.0].max.to_i]
  end
end
# Definimos nuestra clase Image que contendra los datos de todos los pixeles
# (colores) de nuestra imagen y tiene un método para guardarse en formato PPM.
class Image
  # Acceso de lectura y escritura a todas las propiedades.
  attr_accessor :data, :width, :height
  # El constructor recibe ancho y alto de la imagen e inicializa un arreglo de
  # tamaño ancho*alto para almacenar la totalidad de los pixeles de la imagen.
  # Cada campo de nuestro arreglo será un objeto de la clase Color.
  # Guardamos los valores de ancho y alto pues nos serviran para escribir la
  # imagen final en PPM.
  def initialize(width, height)
    @width, @height = width, height
    @data = Array.new()
  end
  # Este método recibe una ruta/nombre de archivo y crea una imágen en formato
  # PPM en la ruta indicada.
  def to_PPM(filename)
    # Basado en la especificación del formato PPM
    File.open(filename+'.ppm', 'w') do |f|
      f.puts("P3") #componentes por pixel
      f.puts("#{@width} #{@height}") #ancho y alto
      f.puts("255") #valores posibles por componente
      # Escribimos cada renglón de valores consecutivamente separados por espacios.
      # R G B R G B R G B ... y al final un salto de línea
      (0...@height).each do |y|
```

```
        # Aqui iteramos sobre cada renglón tomamos el pedazo del arreglo con todos
        # Los pixeles, mappeamos la funcion to_s a cada uno y los juntamos (join)
        # con un espacio entre ellos.
        f.puts(@data[y*@width...y*@width+@width].map{|c| c.to_s}.join(" "))
      end
    end
  end
end
# La clase Material es donde almacenaremos todas las propiedades del objeto
class Material
  # Acceso de sólo lectura a las propiedades
  attr_reader :color, :diffuse, :specular, :shininess, :reflect, :transmit, :ior
  # Constructor de la clase, por el momento sólo usaremos color, posteriormente
  # se usaran (shaders) las demás propiedades de la clase como son los componentes
  # difuso, especular, brillo, reflexion, transmitividad, IOR (índice de refraccion).
  # https://en.wikipedia.org/wiki/Phong_reflection_model
  def initialize(color, diffuse=0.0, specular=0.0, shininess=0.0, reflect=0.0, transmit=0.0, ior=0.0)
    @color = color
    @diffuse = diffuse
    @specular = specular
    @shininess = shininess
    @reflect = reflect
    @transmit = transmit
    @ior = ior
  end
end
# De la Práctica 5
# Clase de un rayo, este tiene un origen y una dirección
# Además debe de poder almacenar la distancia de intersección
# cuando encuentra una colisión, así como la referencia del objeto
# con el que colisiono.
class Ray
  attr_accessor :origin, :direction, :distance, :object
  # Constructor de la clase, sólo definimos el origen y la
  # dirección del rayo, la distancia inicial será la distancia
```

```
  # máxima (horizonte) de nuestra escena, y el objeto de colisión
  # es nulo cuando se crea el rayo.
  # La constante MAX_DISTANCE se define previamente a nivel global
  def initialize(origin, direction)
    @origin = origin
    @direction = direction
    @distance = MAX_DISTANCE
    @object = nil
  end
end
# Práctica 3
# Definimos una clase genérica de objeto de la cual heredaran
# nuestras primitivas y todos los objetos virtuales.
# Todos los objetos deberan implementar el método 'intersect(ray)'
# con sus propias funciones de colisión, así como 'get_normal(point)'
# para calcular la normal en el punto de colisión.
# Nota: La función intersect es de la practica 5
class Object3D
  attr_reader :type, :material
  # Todos los objetos tienen un tipo ("plano", "esfera", etc...)
  # Y un material, que correspondera a una referencia en la lista
  # de materiales.
  def initialize(type, material)
    @type = type
    @material = material
  end
end
# Clase para la esfera nuestra primera primitiva.}
# La esfera como primitiva tiene una resolución infinita lo que significa
# que tiene infinitas caras, para definirla sólo requerimos de la posición
# o centro de la esfera y de su radio.
class Sphere < Object3D
  attr_reader :center, :radius
  # Constructor de la esfera, se requiere su material como en cualquier
  # objeto, así como su centro (vector) y su radio (escalar)
```

```
def initialize(material, center, radius)
  super('sphere', material)
  @center = center
  @radius = radius
end
# Sirve para calcular la normal en un punto de la esfera.
# Es el vector que va desde el centro hasta el punto, normalizado.
def get_normal(point)
  return (point - @center).normalize!
end
# De la práctica 5
# La función de intersección del plano, revisa si hay colisiones entre
# 'ray' (el rayo que recibe) y el plano devuelve true o false dependiendo
# y en caso de ser true asigna valores al rayo.
def intersect(ray)
  # Calculamos un vector que vaya desde el centro de nuestra esfera hasta
  # el origen del rayo.
  sph_ray = @center - ray.origin
  # Con el producto punto podemos calcular la distancia que hay entre el
  # rayo y la esfera. Si le restamos el radio de la esfera y esa distancia
  # es mayor a la que ya tenia el rayo, la esfera no es el objeto más cercano
  # así que podemos ignorarla.
  v = sph_ray.dot(ray.direction)
  return false if v - @radius > ray.distance
  # Si es menor, resolvemos la ecuación quadrática para revisar colisión
  # entre el rayo y el plano de corte de la esfera.
  inter = @radius**2 + v**2 - sph_ray.x**2 - sph_ray.y**2 - sph_ray.z**2
  # Si es negativa estamos viendo una cara interior de la esfera, ya sea
  # la parte trasera de la esfera (segunda colisión) o bien la camara se
  # encuentra dentro de la esfera.
  return false if inter < 0.0
  # Calculamos la distancia del rayo al plano en que el rayo colisiona
  # con la esfera pues sabemos que hay intersección.
  inter = v - Math.sqrt(inter)
  # Si la distancia de intersección es negativa, la colisión se encuentra
```

```
    # detrás de la cámara, así que la ignoramos. Del mismo modo si la
    # distancia de colisión es mayor a una que el rayo haya encontrado antes
    # la descartamos pues no es el objeto más cercano.
    return false if inter < 0.0
    return false if inter > ray.distance
    # Si nada de lo anterior se cumple, la colisión es la más cercana hasta
    # ahora, así que guardamos en el rayo la distancia y la referencia del
    # objeto con el que choco.
    ray.distance = inter
    ray.object = self
    # Devolvemos true pues encontramos una colisión válida.
    return true
  end
end
# Clase para el plano, segunda primitiva.
# En el caso de un Plano como primitiva se considera que el plano es
# infinito como se define matemáticamente. Lo único que se requiere
# para definirlo es su normal, y la distancia a la que el plano está
# del origen.
class Plane < Object3D
  attr_reader :normal, :distance
  # Inicializamos el plano con su material, su normal y su distancia.
  def initialize(material, normal, distance)
    super('plane', material)
    @normal = normal
    @distance = distance
  end
  # La normal de un plano en cualquier punto de este siempre es la normal
  # del plano, así que simplemente devolvemos el vector almacenado.
  def get_normal(point)
    return @normal
  end
  # De la práctica 5
  # La función de intersección del plano, revisa si hay colisiones entre
  # 'ray' (el rayo que recibe) y el plano devuelve true o false dependiendo
```

```
  # y en caso de ser true asigna valores al rayo.
  def intersect(ray)
    # Calculamos el producto punto entre la normal del plano y la dirección
    # del rayo, un rayo siempre colisionará con un plano a menos de que sean
    # paralelos.
    v = @normal.dot(ray.direction)
    # Si el producto punto es 0 significa que el ángulo entre el rayo y la
    # normal del plano es de 90°. Lo cual implica que el rayo y el plano
    # son paralelos y nunca colisionarán así que regresamos false.
    return false if v == 0.0
    # Calculamos la distancia de intersección entre el origen del rayo y la
    # superficie del plano.
    inter = -(@normal.dot(ray.origin) + @distance) / v
    # Si la distancia de intersección es negativa, la colisión se encuentra
    # detrás de la cámara, así que la ignoramos. Del mismo modo si la
    # distancia de colisión es mayor a una que el rayo haya encontrado antes
    # la descartamos pues no es el objeto más cercano.
    return false if inter < 0.0
    return false if inter > ray.distance
    # Si nada de lo anterior se cumple, la colisión es la más cercana hasta
    # ahora, así que guardamos en el rayo la distancia y la referencia del
    # objeto con el que choco.
    ray.distance = inter
    ray.object = self
    # Devolvemos true pues encontramos una colisión válida.
    return true
  end
end
# Práctica 6
# Clase para almacenar la información de las fuentes de luz
class Light
  # Acceso de sólo lectura a las variables de clase que obtendremos de las
  # fuentes de luz, su posición y color en el caso de las fuentes puntuales,
  # y sólo el color en el caso de las fuentes de tipo ambiental
  attr_reader :position, :color, :type
```

```
  # El contructor de nuestra clase sólo almacena los datos necesarios
  def initialize(type, position, color)
    @type = type
    @position = position
    @color = color
  end
end
# Práctica 4
# Clase para cargar la escena y guardar todos los objetos,
# luces, materiales, imagen, etc...
# Todo se carga desde un archivo de texto con la definición
# de la escena, es importante que esta clase vaya implementando
# los conceptos adicionales que se vayan utilizando.
class Scene
  # Acceso de sólo lectura a las propiedades necesarias
  attr_reader :image, :vhor, :vver, :vp, :cam_pos, :objects, :materials, :lights, :depth, :oversampling
  # El constructor de la clase recibe el nombre del archivo de
  # la escena y carga todo su contenido en las variables que
  # serán utilizadas por el ray tracer
  def initialize(filename)
    @objects = Array.new()
    @materials = Array.new()
    @lights = Array.new() # Práctica 6
    # Valores predeterminados
    @image = Image.new(320, 240) # Tamaño 320x240
    # FoV (Field of view) o campo de visión es el ángulo de
    # apertura de la cámara.
    @fov = 60
    @depth = 3 # profundidad de trazado
    @oversampling = 1 # Sobre muestreo (1 = no oversampling)
    # Abrimos y leemos el archivo linea por linea
    File.open(filename).each do |data|
      # Separamos cada linea (split) por espacios (" ") y
      # quitamos los espacios adicionales que pudieran
      # haber en el archivo (strip)
```

```
    line = data.split(" ").map{|i| i.strip}
    # Seleccionamos el primer elemento line[0] y revisamos los
    # posibles casos para cargarlo a memoria
    case line[0]
      when "#"  # comentario en el archivo, lo ignoramos.
        next
      when "field_of_view" # campo de vision
        @fov = line[1].to_f
      when "oversampling"
        @oversampling = line[1].to_i
      when "depth"
        @depth = line[1].to_i
      when "image_size" # tamaño de la imagen
        @image.width = line[1].to_i
        @image.height = line[2].to_i
      when "camera_position" # vector de posición de la camara
        @cam_pos = parse_vector(line[1..3])
      when "camera_look" # vector de hacia donde mira la camara
        @cam_look = parse_vector(line[1..3])
      when "camera_up" # vector de hacia donde es arriba para la camara
        @cam_up = parse_vector(line[1..3])
      when "material" # color(r, g, b), diffuse, specular, shininess, reflect, transmit, ior
        @materials << parse_material(line)
      when "plane" # material, normal(x, y, z), distance
        @objects << Plane.new(line[1].to_i, parse_vector(line[2..4]), line[5].to_f)
      when "sphere" # material, center(x, y, z), radius
        @objects << Sphere.new(line[1].to_i, parse_vector(line[2..4]), line[5].to_f)
      when "light" # Práctica 6 - type, position(x, y, z), color(r, g, b)
        @lights << Light.new(line[1], parse_vector(line[2..4]), parse_color(line[5..7]))
      else # desconocido
        next
    end #end case
  end # end File
  # Auto calculo del vector up si no existe (experimental)
  if @cam_up.nil?
```

```
    @cam_up = @cam_look.cross(Vector3.new(0.0,0.0,1.0)).cross(@cam_look)
  end
  # Continuamos calculando las variables una vez parseado el archivo
  # Grid es la cuadricula a través de la cual trazaremos los rayos.
  # La cuadricula se multiplica por el oversampling cuando este implementado
  @grid_width = @image.width * oversampling
  @grid_height = @image.height * oversampling
  # Hacemos cálculos sobre los vectores de la cámara para trazar la piramide
  # formada por la cuadrícula y el origen de la cámara.
  @look = @cam_look - @cam_pos
  # vhor y vver son los vectores que definen nuestra vista horizontal y vertical
  # y los normalizamos, ambos se calculan usando producto cruz para obtener
  # los vectores perpendiculares.
  @vhor = @look.cross(@cam_up)
  @vhor.normalize!
  @vver = @look.cross(@vhor)
  @vver.normalize!
  # La constante PI_OVER_180 se define previamente 3.1415/180
  fl = @grid_width / (2 * Math.tan((0.5 * @fov) * PI_OVER_180))
  # Copiamos look para normalizarlo
  vp = @look
  vp.normalize!
  vp.x = vp.x * fl - 0.5 * (@grid_width * @vhor.x + @grid_height * @vver.x)
  vp.y = vp.y * fl - 0.5 * (@grid_width * @vhor.y + @grid_height * @vver.y)
  vp.z = vp.z * fl - 0.5 * (@grid_width * @vhor.z + @grid_height * @vver.z)
  @vp = vp
  # stats
  #puts "Objects #{@objects.length}"
  #puts "Materials #{@materials.length}"
  #puts "Camera #{@cam_pos}, #{@cam_look}, #{@cam_up}"
end # end initialize
# Funciones auxiliares para parsear el archivo devuelven los objetos convirtiendo
# los valores a flotantes para su contrucción
def parse_vector(line)
  return Vector3.new(line[0].to_f, line[1].to_f, line[2].to_f)
```

```
  end
  def parse_color(line)
    return Color.new(line[0].to_f, line[1].to_f, line[2].to_f)
  end
  def parse_material(line)
    f = line[4..-1].map{|l| l.to_f}
    return Material.new(parse_color(line[1..3]), f[0], f[1], f[2], f[3], f[4], f[5])
  end
end
# Práctica 5
# Clase Raytracer, esta contendrá todo lo necesario para generar
# la imagen final a partir de la descripción de escena.
# Los métodos importantes son trace(ray) que se dedica a trazar rayos
# y render_pixel(x, y) que es la que renderiza el pixel en esas coordenadas
# de la imagen. La función render es el ciclo principal de renderizado.
class Raytracer
  # El constructor es igual al de la escena pues sólo se encarga de
  # construirla y empezar el proceso.
  def initialize(filename)
    # Guardamos el nombre para utilizarlo como nombre de la imagen
    @filename = filename
    @scene = Scene.new(filename)
  end
  # Práctica 6
  # Método para calcular sombras, recibe el rayo generado a partir del
  # punto de colisión y el objeto con el que colisionó. regresa la
  # la atenuación de color [0.0. 1.0] o sombra en ese punto.
  def calculate_shadow(ray, colision_object)
    shadow = 1.0 # Significa que originalmente no hay sombra
    @scene.objects.each do |obj|
      # Si hay colisión y el objeto es distinto al actual.
      if obj.intersect(ray) and obj != colision_object
        # Mltiplicamos la sombra por la opacidad del objeto
        # colisionado. Objectos traslucidos hacen menos
        # sombra detrás de ellos.
```

```
        shadow *= @scene.materials[obj.material].transmit
      end
    end
    return shadow
  end

  # La función trace será la función que emita los rayos y revise las
  # colisiones contra los objetos dela escena. Posteriormente se
  # convertirá en una función recursiva para el trazado de rayos
  # y recibirá el parametro 'depth' o la profundidad de trazado
  # para poder detener el proceso. Devuelve el color del objeto.
  def trace(ray, depth)
    # Creamos un color nuevo para ir almacenando el resultado
    c = Color.new()
    # Iteramos sobre todos los objetos de la escena y probamos la
    # intersección del rayo con cada uno, hay que notar que en caso
    # de colisión el rayo tendrá la referencia al objeto más cercano
    # y la distancia de colisión con el objeto.
    @scene.objects.each do |obj|
      obj.intersect(ray)
    end
    # Revisamos si hubo interseccion para devolver el color del objeto
    if !ray.object.nil?
      # Nuestro objeto guarda el indice de la lista de materiales
      # buscamos en la lista el color del material para devolverlo.
      # return @scene.materials[ray.object.material].color
# ***** Proyecto 2 *****
# *** Práctica 6 *** - revisaremos las luces y calcularemos sombra.
      # Primero extraemos el material del objeto colisionado
      mat = @scene.materials[ray.object.material]
      #c = mat.color # Practica 6
      # Cálculamos el punto de intersección con el objeto
      inter_point = ray.origin + ray.direction * ray.distance
      # Por último la normal en el punto de colisión
      inter_normal = ray.object.get_normal(inter_point)
```

```
      # Calculamos un vector tenga sentido opuesto a la dirección del rayo
      # (back origin) de regreso al origen y lo normalizamos
      back_origin = ray.direction * -1.0
      back_origin.normalize!
      # Ahora iteramos sobre todas nuestras fuentes de luz
      @scene.lights.each do |light|
        # Si la luz es de tipo ambiental, simplemente sumamos
        # el color de la luz a nuestro color resultado
        if light.type == "ambient"
          c += light.color
        # Si la luz es de tipo puntual...
        elsif light.type == "point"
          # Caluclamos la el vector que va de la luz hacia el punto
          # de intersección con nuestro objeto para ver si hay sombra
          light_dir = light.position - inter_point
          light_dir.normalize!
          # Creamos un nuevo rayo que va de nuestro punrto de interseccion
          # hacia la fuente de luz para revisar colisiones con otros objetos
          light_ray = Ray.new(inter_point, light_dir)
          # Y calculamos la sombra...
          shadow = calculate_shadow(light_ray, ray.object)
          #return c *= shadow # Practica 6
# *** Práctica 7 *** - Sombreado de superficies, componente difuso y especular
          # Calculamos el producto punto entre la normal y la luz
          nl = inter_normal.dot(light_dir)
          # Si no son perpendiculares procedemos a calcular sus componentes
          # difusa y especular. Si fueran perpendiculares quiere decir que
          # la luz es tangente al punto de intersección y no añade al
          # sombreado de la superficie en ese punto.
          if nl > 0.0
            if mat.diffuse > 0.0 #----- Componente difusa
              # El color del componente difuso es igual al color de la luz
              # multiplicado por el coeficiente difuso y por el coseno del
              # ángulo entre la normal y la luz en el punto de intersección
              # (producto punto)
```

```
            diffuse_color = light.color * mat.diffuse * nl
            # Despues agregamos el color del material y la sombra a cada
            # componente RGB
            diffuse_color.r *= mat.color.r * shadow
            diffuse_color.g *= mat.color.g * shadow
            diffuse_color.b *= mat.color.b * shadow
            # Y lo sumamos a nuestro color resultado
            c += diffuse_color
          end # end diffuse
          if mat.specular > 0.0 #----- Componente especular
            # calcular el componente especular - Phong
            # Primero calculamos el vector reflejado en la superficie
            r = (inter_normal * 2 * nl) - light_dir
            spec = back_origin.dot(r)
            # Si el producto punto es 0.0 son perpendiculares lo que implica
            # que el punto de colisión es tangente a la iluminación, lo ignoramos
            if spec > 0.0
              # Calculamos la componente especular con la dureza o brillo del material
              spec = mat.specular * spec**mat.shininess
              # Y creamos nuestro color especular agregando la sombra
              specular_color = light.color * spec * shadow
              # Sumamos este color a nuestro resultado final
              c += specular_color
            end #end spec
          end # end specular
        end # end Práctica 7 (nl)
      end # end light if
    end # end light loop
# *** Práctica 8 **** -  Reflexión y Refracción
    # A partir de aquí el algoritmo se vuelve recursivo y se reutiliza
    # para trazar rayos en los puntos de colisión hasta que lleguemos
    # a nuestro máximo valor de profundidad de trazado.
    # Y un vector en sentido opuesto al rayo (regresa al origen)
    if depth < @scene.depth
      if mat.reflect > 0.0 #----- Reflexión
```

```
    # Producto punto
    t = back_origin.dot(inter_normal)
    # Si es 0.0 los vectores son perpendiculares lo que implica que
    # la reflexión es tangente a la superficie y no importa.
    if t > 0.0
      # Calculamos la dirección que deberá tener el rayo reflejado
      dir_reflect = (inter_normal * 2 * t) - back_origin
      # offset sirve para separar de manera insignificante (SMALL)
      # el punto de colisión de la superficie del objeto para
      # evitar que el rayo reflejado choque con su propia superficie
      offset_inter = inter_point + dir_reflect * SMALL
      # Creamos el nuevo rayo de reflexión a partir del punto de
      # colisión y con la dirección del rayo reflejado en la superficie
      reflection_ray = Ray.new(offset_inter, dir_reflect)
      # Recursivamente trazamos el nuevo rayo de reflexión con nuestra
      # misma función trace e incrementamos la profundidad de trazado.
      # En este caso nuestro resultado final lo multiplicamos por el
      # factor de reflexión del material.
      c += trace(reflection_ray, depth+1.0) * mat.reflect
    end # end t
  end # end reflect
  if mat.transmit > 0.0 #----- Refracción
    # Calculamos el vector incidente
    incident = inter_point - ray.origin
    rn = inter_normal.dot(incident * -1.0)
    incident.normalize!
    if inter_normal.dot(incident) > 0.0
      inter_normal = inter_normal * -1.0
      rn = -rn
      n1 = mat.ior
      n2 = 1.0
    else
      n1 = 1.0
      n2 = mat.ior
    end # end incident
```

```
          if n1 != 0.0 and n2 != 0.0
            par_sqrt = Math.sqrt(1 -(n1*n1/n2*n2)*(1-rn*rn))
            dir_refract = incident + (inter_normal*rn) * (n1/n2) - (inter_normal*par_sqrt)
            offset_inter = inter_point + dir_refract * SMALL
            refraction_ray = Ray.new(offset_inter, dir_refract)
            c += trace(refraction_ray, depth+1.0) * mat.transmit
          end
        end # end transmit
      end # end depth
    end # end object loop
    # Si no hubo regresamos el color deafult (negro)
    return c
  end
  # La función render_pixel(x, y) crea los rayos a partir de la cámara,
  # manda llamar trace y almacena el color final en nuestra imagen en
  # las coordenadas x, y
  def render_pixel(x, y)
# *** Práctica 9 *** - Oversampling
    c = Color.new()
    x *= @scene.oversampling
    y *= @scene.oversampling

    @scene.oversampling.times do
      @scene.oversampling.times do
        # Cálculamos la dirección del rayo interpolando en la cuadricula
        # de la cámara.
        direction = Vector3.new()
        direction.x = x * @scene.vhor.x + y * @scene.vver.x + @scene.vp.x
        direction.y = x * @scene.vhor.y + y * @scene.vver.y + @scene.vp.y
        direction.z = x * @scene.vhor.z + y * @scene.vver.z + @scene.vp.z
        direction.normalize!
        # El origen del rayo siempre será la posición de la cámara.
        ray = Ray.new(@scene.cam_pos, direction)
        c += trace(ray, 1.0)
        y += 1
```

```
      end # end j loop
      x += 1
    end # end i loop
    sqr_os = @scene.oversampling * @scene.oversampling
    c.r /= sqr_os
    c.g /= sqr_os
    c.b /= sqr_os
    # Regresamos el color que trazamos en nuestra imagen
    return c
  end
  # Render es el ciclo principal, donde iteramos sobre toda la imagen
  # y después guardamos el resultado.
  def render()
    (0...@scene.image.height).each do |y|
      print "Rendering line %.3d of %d\r" % [y+1, @scene.image.height]
      (0...@scene.image.width).each do |x|
        # Rendereamos cada pixel
        @scene.image.data[y*@scene.image.width+x] = render_pixel(x, y)
        #print "[", x, ",", y, "] "
      end
    end
    # Guardamos la imagen
    @scene.image.to_PPM(@filename)
  end
end
# Inicializamos el raytracer con una escena y lo empezamos
filename = ARGV[0] || "scene.txt"
puts "Loading " + filename
rt = Raytracer.new(filename)
rt.render()
```