UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria
de Manresa

Treball Final de Grau

Implementació d'un Ray Tracer

Grau en enginyeria de sistemes TIC
Curs 19/20

Autor: Joel Anguera Comas
Director: Sebastià Vila Marta
Data: 09/10/2020
Localitat: Manresa

RESUM

El treball consisteix en la implementació d'un ray tracer, que és un algoritme de renderització per crear una imatge a partir d'un model 3D.

Un ray tracer és un algoritme que es basa en simular la realitat. Mitjançant un sistema de traçat de raigs, determina la posició dels objectes en una escena i calcula la quantitat de llum que els hi arriba per saber-ne el color.

Els raigs es llancen des d'un observador (càmera) cap a l'escena a través del pla de la imatge. Es calculen les interseccions dels raigs amb els diferents objectes de l'escena i la intersecció que queda més propera a l'observador determina quin es l'objecte visible. Des d'aquest punt d'intersecció es llancen més raigs per calcular la quantitat de llum que arriba a l'objecte.

Després de traçar suficients raigs, s'aconsegueix una imatge de l'escena des del punt de vista de l'observador.

ABSTRACT

The aim of this project is to implement a ray tracer. A ray tracer is a render algorithm to create an image out of a 3D model.

A ray tracer is an algorithm which is based on how reality works. It uses a set of rays to get the position of the objects on a scene and then computes the amount of light those objects receive to calculate their color.

The rays are traced from an observer (camera) to the scene through an image plane. Then the rays are tested against the objects to check if there are any ray-object intersections. The object whose intersection is closer to the observer is the one visible for him. From this intersection point more rays are traced through the scene to calculate the amount of light that the point on that object is getting.

When enough rays are traced, you get a clear image of the scene from the viewer's point of view.

ÍNDEX

1. INTRODUCCIÓ

Aquest treball neix amb fins acadèmics com una introducció al món de la computació gràfica, i s’hi tracta una tècnica de tractament gràfic en concret, el Ray Tracing. Els gràfics per ordinador són un tema que tenim molt per mà i que estem molt habituats a veure, però que en general no se’n sap gran cosa. Veiem imatges generades per ordinador constantment en gran part del contingut multimèdia que consumim, però no som conscient de la feina que porta crear-les.

La computació gràfica és una enorme i recent àrea de les ciències de computació. Hi ha moltes tècniques estudiades i desenvolupades per poder traslladar els dissenys en 3D que entenen els ordinadors a models 2D que es poden representar en pantalles o papers. El cas del Ray Tracing, per exemple, fins fa uns anys no era una tècnica de renderització viable, ja que era massa costosa a nivell de càlcul, i actualment s'està començant a implementar com a tècnica de renderització en temps real (es poden renderitzar les imatges tant ràpidament que creen l’efecte de moviment).

Així doncs, al ser un tema recent en el que actualment s’hi està aplicant molta investigació, pot resultar interessant conèixer les bases del seu funcionament.

La implementació de l’algoritme de ray tracing desenvolupada en aquest projecte està enfocada amb fins il·lustratius, de manera que la seva estructura segueix un model didàctic. A més s’ha seguit una estructura que facilita la seva ampliació o la modificació dels seus elements.

1.1. Objectius

Com ja s’ha comentat, l’objectiu d’aquest treball es basa en fins acadèmics per introduir conceptes com la computació gràfica i el funcionament d’un algoritme de ray tracing. Així doncs, els algoritmes, les estructures de dades i tècniques de renderització que s’han fet servir han estat escollides amb cura per facilitar al màxim aquesta explicació.

2. ANTECEDENTS

En aquest apartat es presenten conceptes bàsics per poder entendre correctament tota l'explicació d'aquest projecte. Dir també que aquest projecte està escrit en el llenguatge de programació C i que al llarg de la memòria es fan vàries referencies a trossos del codi escrit i a diferents tipus de dades d'aquest llenguatge.

També s'ha fet ús de la geometria i altres conceptes matemàtics que pot ser que no estiguin explicats del tot i que es donin per sabuts.

2.1. Renderització

La renderització és el procés de portar una escena o model en tres dimensions a una imatge en dos dimensions, que es pot veure per exemple en un monitor.

Aquesta escena està composta d'objectes descrits en un llenguatge o estructura de dades i poden tenir diferents característiques: la seva posició, textura, transparència... Aquesta informació és processada per un programa de renderitzat i ens retorna una imatge en dos dimensions.

La renderització es fa servir en programes de disseny 3D (ja sigui en l'arquitectura, indústria automobilística...), videojocs, efectes especials de sèries i pel·lícules, en pel·lícules d'animació 3D, etc. Cada un d'aquests àmbits pot tenir les seves necessitats i fer ús de diferents mètodes de renderització, per exemple en un videojoc el renderitzat s'ha de fer en temps real, en canvi en una pel·lícula d'animació 3D no és necessari.

El procés de renderització es pot separar en dos parts: visibilitat i ombrejat. En la part de visibilitat s'ha de descobrir quins objectes de l'escena són visibles des de la perspectiva escollida, i en la part d'ombrejat s'ha de calcular el color dels objectes visibles.

2.2. Ray tracing

El ray tracing és un mètode de renderització que es basa en reproduir o simular la realitat: fa servir principis físics per modelar la interacció de la llum i la matèria.

La brillantor i color amb la que veiem els objectes es degut a la interacció de rajos de llum amb aquests objectes. Els rajos de llum estan formats per fotons (partícules electromagnètiques) i són emesos per vàries fonts de llum, sent una de les mes destacables, el sol. Quan un grup de fotons col·lisiona amb un un objecte poden passar tres coses: poden ser absorbits, reflectits o transmesos. El percentatge de fotons reflectits, refractats o transmesos depèn de cada material i majoritàriament és el que dicta com veiem l'objecte.

Els grecs van desenvolupar una teoria de la visió en la que deien que els objectes eren visibles gràcies a uns raigs de llum sortits dels ulls. Temps després, va ser un científic àrab: Ibn al-

Hàytham (965-1040) qui va explicar per primer cop que veiem els objectes gràcies als rajos del sol.

Fer renderitzacions basades en els principis físics de la realitat pot semblar una forma òbvia de renderització, però aquesta tècnica no es va començar a fer servir productivament fins uns quants anys després del seu descobriment.

Amb la publicació de l’article "An Improved Illumination Model for Shaded Display" al 1980 per Turner Whitted, on va introduir la idea de fer servir el ray tracing per reproduir la il·luminació general d’una escena, va obrir la porta a la reproducció de manera realista de la distribució de la llum en les escenes. Aleshores la idea de renderització fent ús de principis físics es va començar a plantejar seriosament. Van començar a sortir més articles i estudis sobre el tema, però degut a la seva complexitat de càlcul i que les màquines d’aquells temps eren molt cares i poc potents, no era un mètode viable de renderització. Així doncs, l’ús del ray tracing no es va començar a estendre fins el 1990-2000.

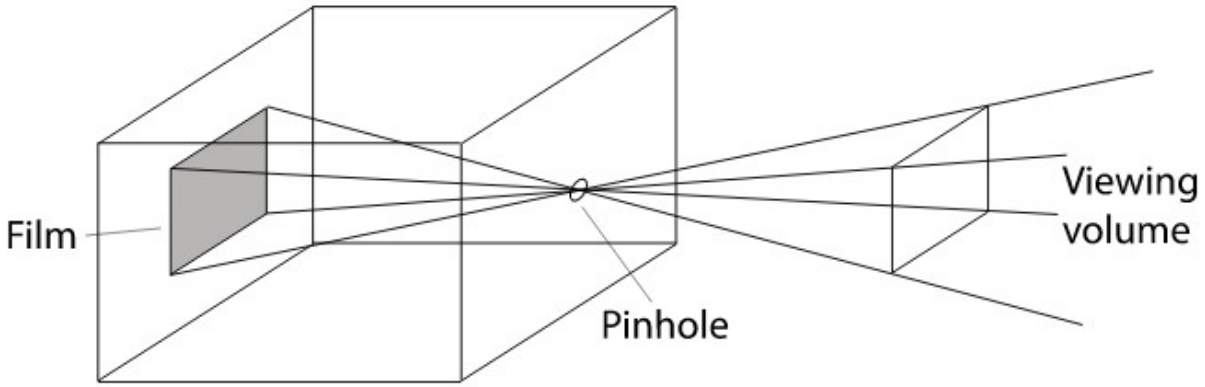
2.3. Structs

Els `structs` són un tipus de dada del llenguatge de programació C. En aquest projecte, és un tipus de dada que s’ha utilitzat molt ja que es pot fer servir com a argument en una funció, pot ser retornat per una funció, ens permet agrupar varis tipus de dades... Totes aquestes propietats ens són de gran utilitat per la implementació d’aquest projecte.

3. FUNCIONAMENT D'UN RAY TRACER

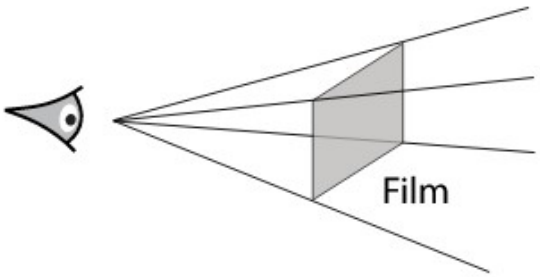
Casi tothom alguna vegada ha fet servir una càmera de fotos i està familiaritzat amb el seu funcionament: apuntes amb l'objectiu a algun lloc del qual en vulguis guardar una imatge i després d'apretar el botó o polsar sobre una pantalla la imatge queda guardada sobre una pel·lícula o sensor electrònic. Un dels dispositius més simples per fer una fotografia és la càmera estenopeica.

Les càmeres estenopeiques consisteixen en una caixa fosca amb un forat en un costat (Il·lustració 3.1). Quan es destapa el forat, la llum hi entra a través i acaba cremant un tros de paper fotogràfic situat a l'altre costat de la caixa. Aquests tipus de càmeres necessiten uns temps d'exposició molt elevats perquè entri suficient llum per formar la imatge.



Il·lustració 3.1: Càmera estenopeica.

Una altre manera de pensar sobre la càmera estenopeica és situar el paper fotogràfic davant del forat però a la mateixa distància. D'aquesta manera obtenim la mateixa visió de l'escena que en el cas anterior. Òbviament, aquesta no és una manera viable de construir una càmera real però per fer una simulació ens serveix. Quan el paper fotogràfic (o pla de la imatge) el situem davant del forat, aquest s'acostuma a anomenar *ull*.



Il·lustració 3.2: Quan simulem una càmera estenopeica situem el paper fotogràfic davant del forat, i el forat passa a ser anomenat ull.

Aleshores s'ha de determinar quin color capta la càmera per a cada punt de la imatge. En la versió original de la càmera estenopeica es veu clar que els únics raigs de llum que ens interessen són els que passen pel forat i acaben impactant contra el paper fotogràfic. En aquesta nova versió on situem el film davant de l'ull, els raigs que ens interessen son els que travessen el film i acaben impactant contra l'ull.

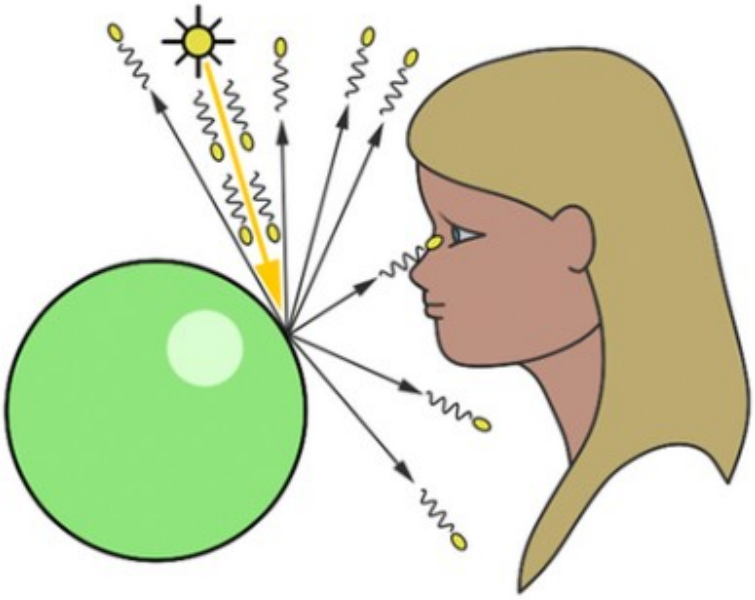
La idea de la càmera estenopeica ens serveix com a símil per explicar el funcionament d'un ray tracer. El que acaba creant la imatge en una càmera estenopeica són els raigs de llum que travessen el paper fotogràfic i l'acaben cremant de manera que es forma una imatge.

El que es fa en un ray tracer es simular la segona versió que s'ha vist de càmera estenopeica (il·lustració 3.2). S'han de simular els raigs de llum i mirar de quin color són els que passen pel pla de la imatge.

Per fer aquesta simulació hi ha dos estratègies diferents, que es veuran a continuació.

3.1. Forward tracing

El «forward tracing» és una estratègia que es basa en seguir els raigs de llum des de que surten de la font de llum fins que arriben a l'ull passant a través del paper fotogràfic. Això és el que passa a la realitat: el sol, per exemple, emet raigs de llum en totes direccions. Aquests raigs reboten en els objectes, i una porció molt petita ens acaba arribant als ulls. Aquests raigs que ens arriben als ulls son la raó per la qual veiem el món que ens envolta.



Il·lustració 3.3: Molts fotons són reflexats per l'esfera verda, però només un arriba a la superfície d el'ull.

Aquesta estratègia presenta un problema però. Del sol, per exemple, ens n'arriben innumerables raigs de llum, que al impactar contra una superfície en surten més raigs reflectits en totes direccions, i només una porció molt petita d'aquests raigs ens acaba arribant als ulls. Per tant,

s'haurien de generar innumerables raigs de llum per tal de que suficients raigs de llum arribessin arribant a l'ull com per poder formar una imatge.

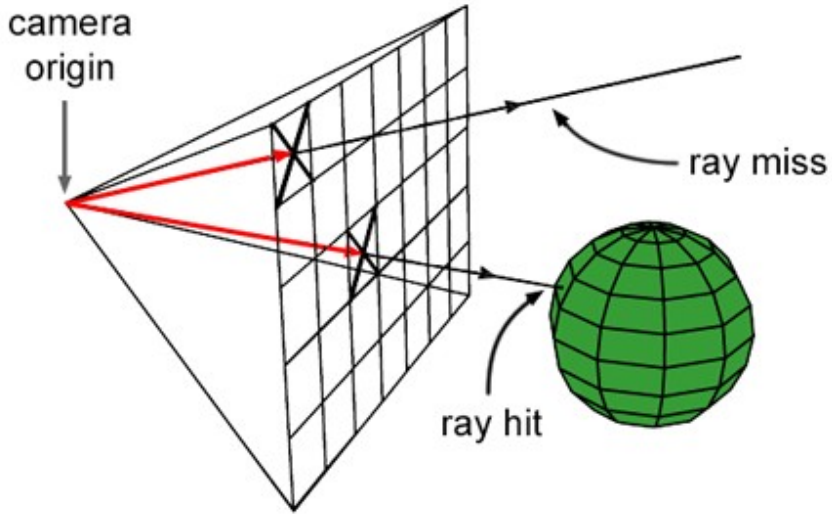
Aquesta estratègia doncs, no resulta satisfactòria.

3.2. Backward tracing

L'altre estratègia consisteix en traçar els raigs en direcció contrària: des de l'ull cap a l'escena. Aquesta estratègia és una bona solució al problema que presentava el «forward tracing», ja que només és simulen els raigs que segur que acaben arribant a l'ull.

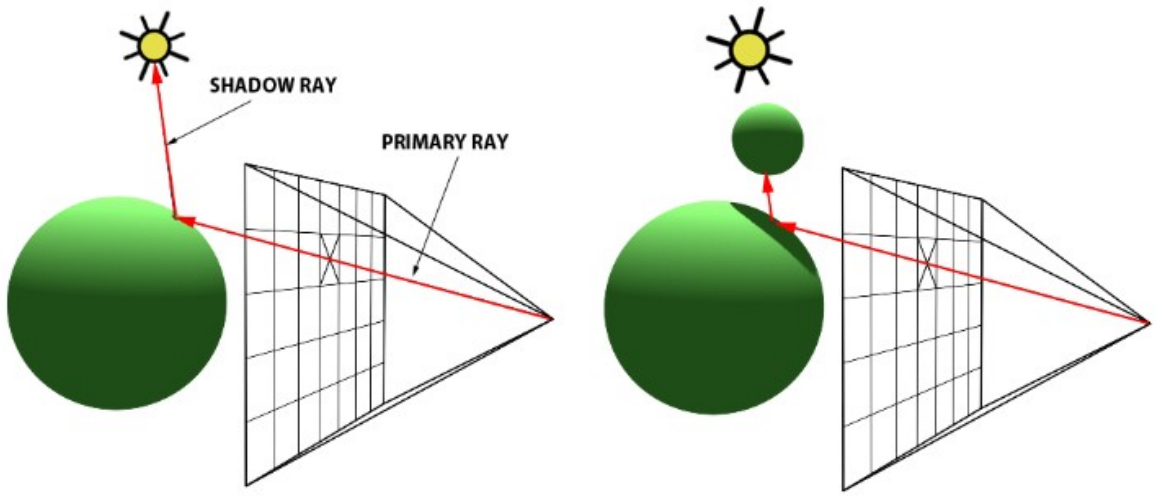
Una imatge digital està formada per una mena de quadrícula de píxels, que són parts molt petites de la imatge. Cada un d'aquests píxels té un color diferent, i el conjunt d'aquests píxels és el que acaba formant la imatge.

El que es fa en un algoritme de ray tracing doncs, és substituir el paper fotogràfic que havíem vist de la càmera estenopeica (il·lustració 3.2) per aquesta quadrícula de píxels i es traça un raig per cada un d'aquests píxels que determinarà quina part de l'escena veu cada píxel. Aquests raigs que surten de l'ull o càmera i que van cap a l'escena s'anomenen raigs primaris.



Il·lustració 3.4: Un raig pot impactar o fallar contra les figures de l'escena.

Una vegada hem generat aquest raig primari, comprovem si col·lisiona per cada un dels objectes de l'escena. Pot arribar a passar que el raig col·lisioni amb més d'un objecte, en aquest cas ens quedem amb l'objecte el qual el punt d'intersecció amb el raig està més a prop de l'ull. Aleshores es traça un segon raig que va des del punt d'intersecció cap a la font llum («*shadow ray*») per saber si en aquest punt li arriba la llum. Si aquest «shadow ray» es troba algun obstacle abans d'arribar a la font de llum significarà que al punt d'intersecció no li arriba la llum (il·lustració 3.5).



Il·lustració 3.5: Disparem el "shadow ray" des del punt de col·lisió per saber si en aquest punt li arriba la llum.

Aquesta és la base d’un algoritme de ray tracing. A partir d’aquí es pot anar ampliant amb reflexions, refraccions, oclusió ambiental i altres tècniques per aconseguir generar imatges cada vegada més realistes. Al incorporar noves tècniques doncs, també s’incrementa la complexitat de l’algoritme, de manera que com més ens vulguem aproximar a la realitat més temps tardarem a renderitzar cada imatge.

3.3. Implementació general d’un Ray Tracer

Un algoritme de ray tracing, és en realitat un algoritme bastant simple. Com ja hem vist, es basa en seguir els rajos de llum en una escena mentre van interactuant i rebotant entre els objectes que la componen. Tot i que hi ha moltes maneres d’escriure un ray tracer, tots haurien de tenir mes o menys aquests components:

- *Càmeres:* Una càmera determina com i des d’on s’està veient una escena. Molts sistemes de renderització generen raigs que s’originen a la càmera i són traçats cap a l’escena.
- *Interaccions entre raigs i objectes:* Hem de ser capaços de determinar amb precisió on un determinat raig col·lisiona amb un objecte, com també poder determinar certes propietats de l’objecte en aquest determinat punt, com la normal o el seu material.
- *Fonts de llum:* Sense il·luminació renderitzar una escena no tindria gaire sentit. Un ray tracer ha de modelar la distribució de la llum a través d’una escena, incloent no només la localització dels focus de llum sinó també com distribueixen la llum a través de l’espai.
- *Visibilitat:* Per saber si a un punt d’un objecte li arriba llum hem de saber si hi ha un camí ininterromput entre aquest punt i alguna font de llum. Afortunadament això és fàcil de saber ja que podem construir un raig que vagi des d’aquest objecte cap a la font de llum i mirar el primer objecte amb el que col·lisiona. Si no col·lisiona amb cap objecte és que es tracta d’un punt lluminós.
- *Informació de la superfície:* Cada objecte ha de tenir l’aparença i la forma com hi interactua la llum descrites.
- *Interacció amb la llum indirecte:* La llum pot arribar a un objecte després de rebotar o passar a traves d’un altre, de manera que s’han de traçar múltiples raigs originaris en el punt de col·lisió per tenir en compte aquest efecte.

3.4. Resum del sistema

L’algoritme de ray tracing desenvolupat en aquest treball està escrit en C. Està basat en un sistema d’`structs` en els que s’hi defineixen certes entitats (per exemple `Surface` és l’`struct` en el que es basen totes les figures geomètriques o objectes de l’escena, l’`struct Light` és el mateix però per les llums). Aleshores hi ha funcions que s’encarreguen de determinar de quin tipus de superfície o objecte es tracta i fan la crida a la funció especifica de l’objecte en qüestió. Per exemple la funció encarregada de determinar si hi ha algun obstacle entre un objecte i una font de llum no necessita tenir en compte de quin tipus de superfície es tracta. Aquest model facilita les coses a l’hora d’ampliar el sistema, ja que per afegir una nova figura només s’ha de definir un nou `struct` amb la informació d’aquest nou objecte, crear la funció per determinar les col·lisions amb aquest nou tipus d’objecte i unir-ho tot a les funcions i `structs` base. A part de facilitar l'ampliació del sistema també fa que sigui fàcil d’entendre.

En aquest apartat es mostra el camí que segueix un raig des de que es crea fins que acaba proporcionant un color per un píxel específic de la imatge.

3.4.1. Representació d'una escena

Per crear una imatge primer es necessita una escena. L'escena és el que conté tots els objectes, llums i opcions de la càmera per tal de poder procedir amb la renderització. La definició d'escena es troba en el mòdul `scene`. Aquest mòdul també té funcions per incorporar tant objectes com fonts de llum a la pròpia escena. Abans d'integrar objectes en una escena però, s'han de crear. Els objectes estan definits en el mòdul `surface`, que també conté funcions per la creació d'aquests objectes. El mateix passa amb les fonts de llum, estan definides en el mòdul `light` i també s'hi proporcionen funcions per la seva creació.

L'escena es crea en el mòdul `main`: primer es creen els objectes i fonts de llum i després s'integren a l'escena. L'escena també conté un color de fons preestablert, que és el color que s'ha de veure en cas que en alguna part de la imatge no hi hagi cap objecte que tapi el fons i les opcions de la càmera, que especifiquen coses com el camp de visió i l'alçada i amplada de la imatge que volem renderitzar. La definició d'escena és la següent:

```
typedef struct {
  Array surfaces;
  Array lights;
  Options options;
  Color backgroundColor;
} Scene;
```

Tots els objectes de l'escena, estan representats per un `struct` del tipus `Surface`, que combina dos tipus d'informació: la forma de l'objecte i el seu material, que descriu la seva aparença (per exemple, el seu color o si té un acabat reflectant o mate). Quan són integrats a l'escena, tots els objectes s'acaben agrupant en un `Array` que és un tipus de dada definida en el mòdul `array` i que és un tipus de llista.

Totes les fonts de llum, estan representades en `structs` del tipus `Light`, que especifica el tipus de llum que és, la intensitat de la llum que proporciona i el seu color. Quan s'acaben integrant a l'escena, totes les fonts de llum s'acaben agrupant en un `Array`, que és el mateix que es fa servir per agrupar els objectes.

Una vegada s'ha generat l'escena ja es pot començar amb el procés de renderització.

3.4.2. Generació d'una imatge

Com ja s'ha comentat, un ray tracer renderitza imatges mitjançant l'ús de raigs. Aquests raigs es llancen de la càmera cap a l'escena i es comprova si col·lisionen amb algun dels objectes de l'escena. En cas afirmatiu, s'ha de determinar la quantitat de llum que arriba a l'objecte en el punt de col·lisió i es calcula un color per el determinat píxel de la imatge pel que ha passat el raig. En cas que no es produeixi cap col·lisió, es retorna el color de fons que s'ha especificat en la creació de l'escena.

Cada vegada que obtenim el color d'un píxel, es guarda en un fitxer que acabarà sent la imatge renderitzada.

Hi ha vàries funcions «principals» que participen en el procés que va des de generar un raig fins a retornar un color. Mes o menys tots els ray tracers implementen d'alguna manera o altre aquest seguit de funcions, ja que son les que marquen el funcionament d'un ray tracer.

A continuació s'explica el funcionament i propòsit d'aquestes funcions d'una manera breu. L'ordre de les funcions que s'explicaran segueix el «cicle de vida» d'un raig, des de que es crea fins que retorna un color.

3.4.3. Funció render

Aquesta funció, definida en el mòdul `ppm_render` és l'encarregada de generar els raigs que s'acabaran llançant cap a l'escena. Els raigs tenen un origen i una direcció i els raigs primaris (els que surten de la càmera) venen determinats per les propietats de la càmera que s'ha especificat per renderitzar l'escena: la posició de la càmera i la mida de la imatge que es vol renderitzar.

La funció `render` itera sobre tots els píxels especificats per la càmera i genera un raig per cada un d'ells. Cada vegada que genera un raig fa una crida a la funció `castRay` i li passa el raig generat. Quan `castRay` li retorna el color del píxel el guarda per, finalment poder-lo escriure en un fitxer i generar la imatge.

3.4.4. Funció castRay

`castRay` està definida en el mòdul `rays`, i és la funció encarregada de traçar el raig cap a l'escena, mirar si col·lisiona amb algun objecte i retornar el color corresponent. Per mirar si el raig col·lisiona amb algun objecte fa ús d'una altre funció anomenada trace, que veurem a continuació. Si resulta que el raig col·lisiona amb algun objecte, ha de calcular quanta llum està arribant en el punt de col·lisió de l'objecte. Per fer això ha d'iterar sobre les diferents fonts de llum que hi ha a l'escena i mirar si la seva llum arriba en el punt en qüestió. La millor manera de fer això és crear un nou raig per cada font de llum amb origen el punt de col·lisió i direcció una font de llum. Aleshores es torna a fer ús de la funció trace per mirar si hi ha algun objecte entremig de la font de llum i el punt de col·lisió.

D'aquesta manera es determina la quantitat de llum que arriba en un punt d'un objecte.

3.4.5. Funció trace

La funció `trace`, que està definida en el mòdul `rays`, és l'encarregada de resoldre el problema de visibilitat. Donat un raig amb un origen i una direcció establertes, itera tots els objectes de l'escena i comprova si el raig en qüestió hi col·lisiona. Cada tipus d'objecte però, té una manera diferent de calcular si un raig hi impacta, ja que diferents objectes tenen estructures diferents, i escriure el codi específic per comprovar la interacció raig-objecte de cada objecte en la funció trace és pràcticament inviable.

Per això, s'ha definit una funció encarregada de gestionar el detectat de col·lisions. Cada vegada que trace ha de comprovar si el raig col·lisiona contra un objecte, fa una crida a aquesta funció independentment del tipus d'objecte que es tracta i aquesta funció fa les gestions necessàries i comprova si hi ha col·lisió. Aquesta funció s'anomena `checkIntersection`, i es l'ultima funció «principal» que hi ha.

Podria passar però, que un raig impacti a més d'un objecte (en cas que un objecte estigui situat davant d'un altre des de la perspectiva de la càmera), si aquest fos el cas, l'objecte que ens interessa és el que queda més a prop de l'origen del raig. De manera que la funció trace no només ha de determinar si el raig col·lisiona amb algun objecte sinó que també ha de dir quin ha estat el

que s'ha trobat primer. Per fer això guarda la distància entre l'origen del raig i cada punt de col·lisió i retorna l'objecte que té una «distància de col·lisió» més petita.

3.4.6. Funció checkIntersection

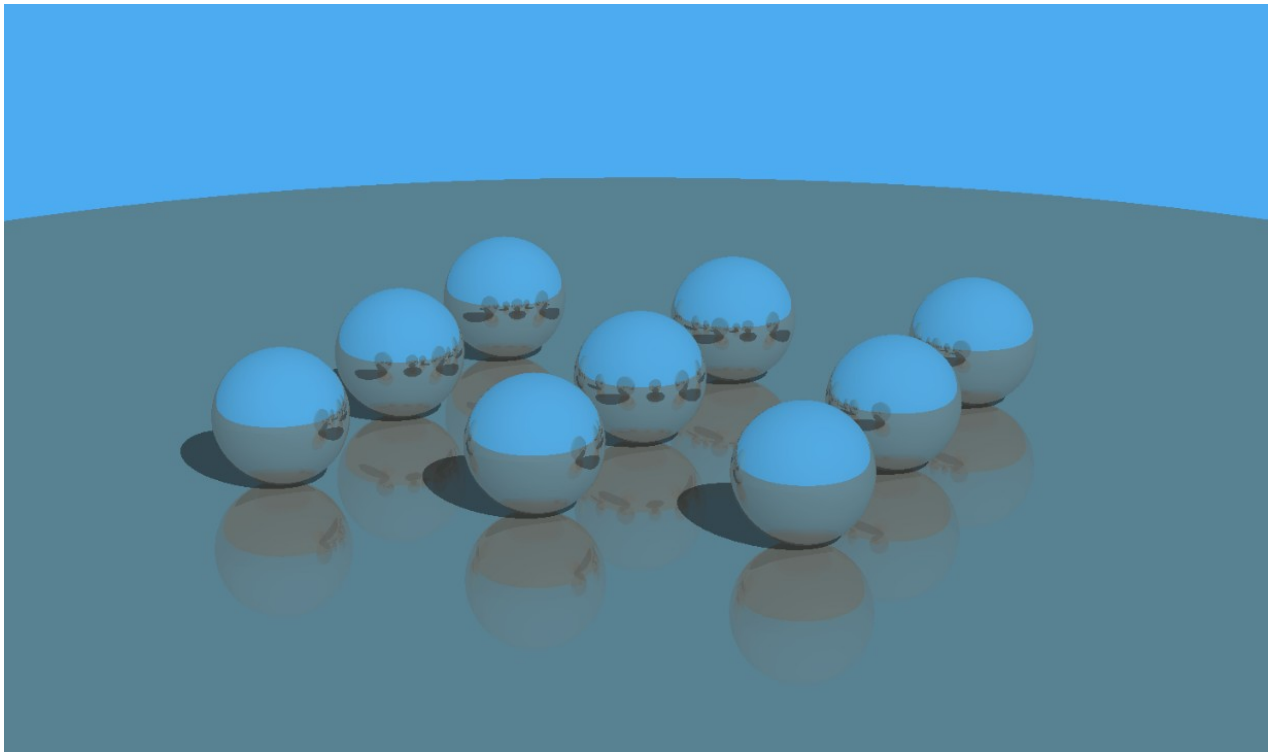
Aquesta funció està definida en el mòdul `rays`, i amb ella es conclouen les funcions «principals» que marquen la vida d'un raig. Aquesta funció, donat un raig i un objecte determinat, ha de comprovar si es creuen en algun moment.

Com veurem en el punt 6 (Objectes i interaccions), cada tipus d'objecte implementat en aquest ray tracer, també té implementada una funció de col·lisió per comprar si un determinat raig hi col·lisiona.

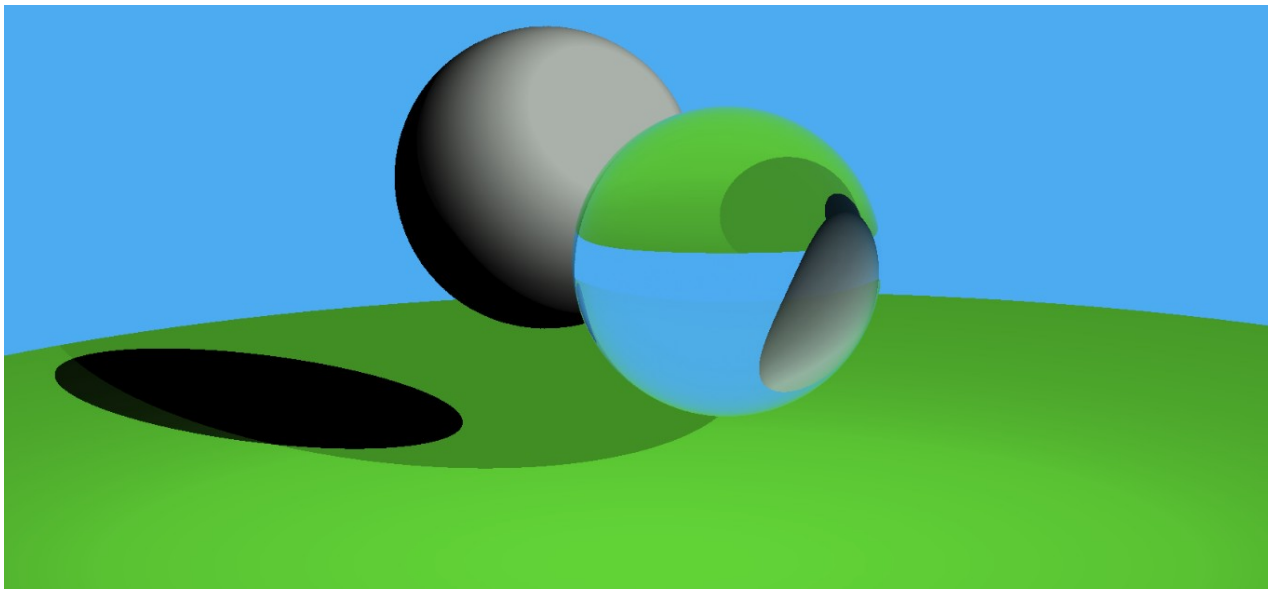
La funció `checkIntersection` doncs, es limita a determinar de quin tipus és l'objecte que se'n vol comprovar la interacció amb el raig i crida la funció especifica de l'objecte en qüestió per fer la comprovació.

3.5. Exemples de renderitzacions

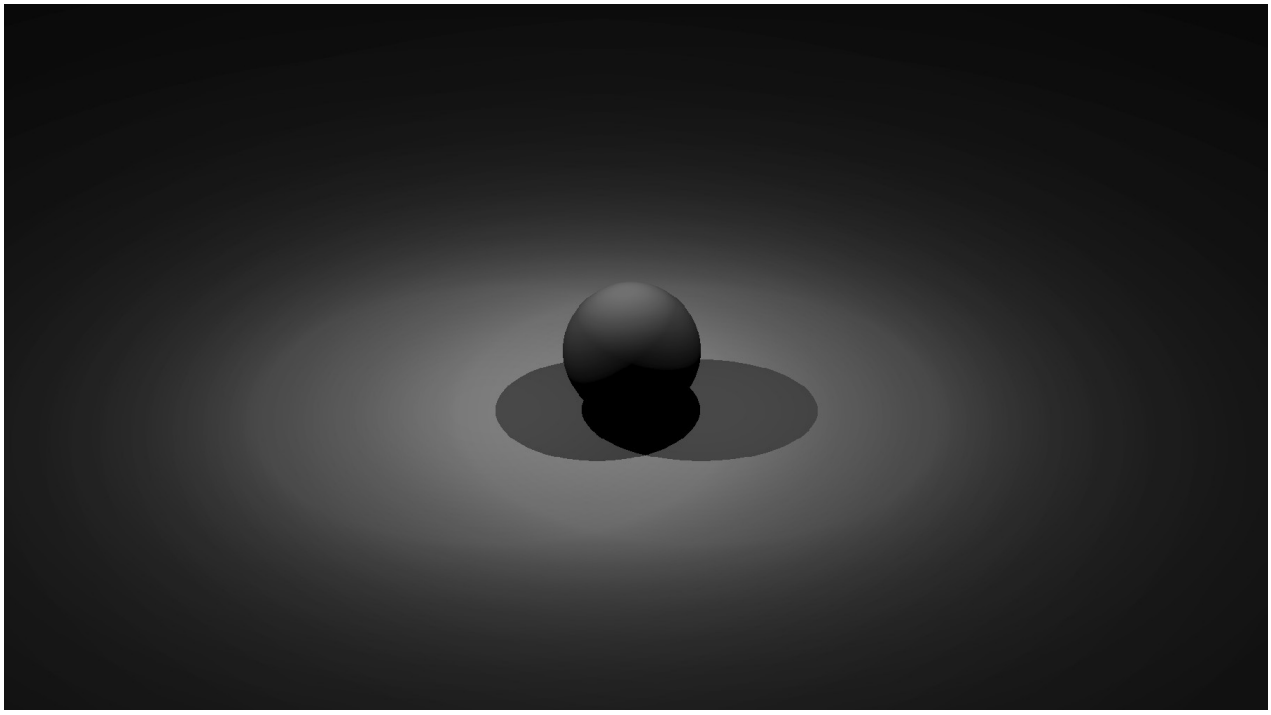
A continuació es deixen unes imatges d'exemple renderitzades amb la versió final del ray tracer implementat en aquest treball. Aquestes imatges inclouen varies figures geomètriques i exemples de reflexions i refraccions.



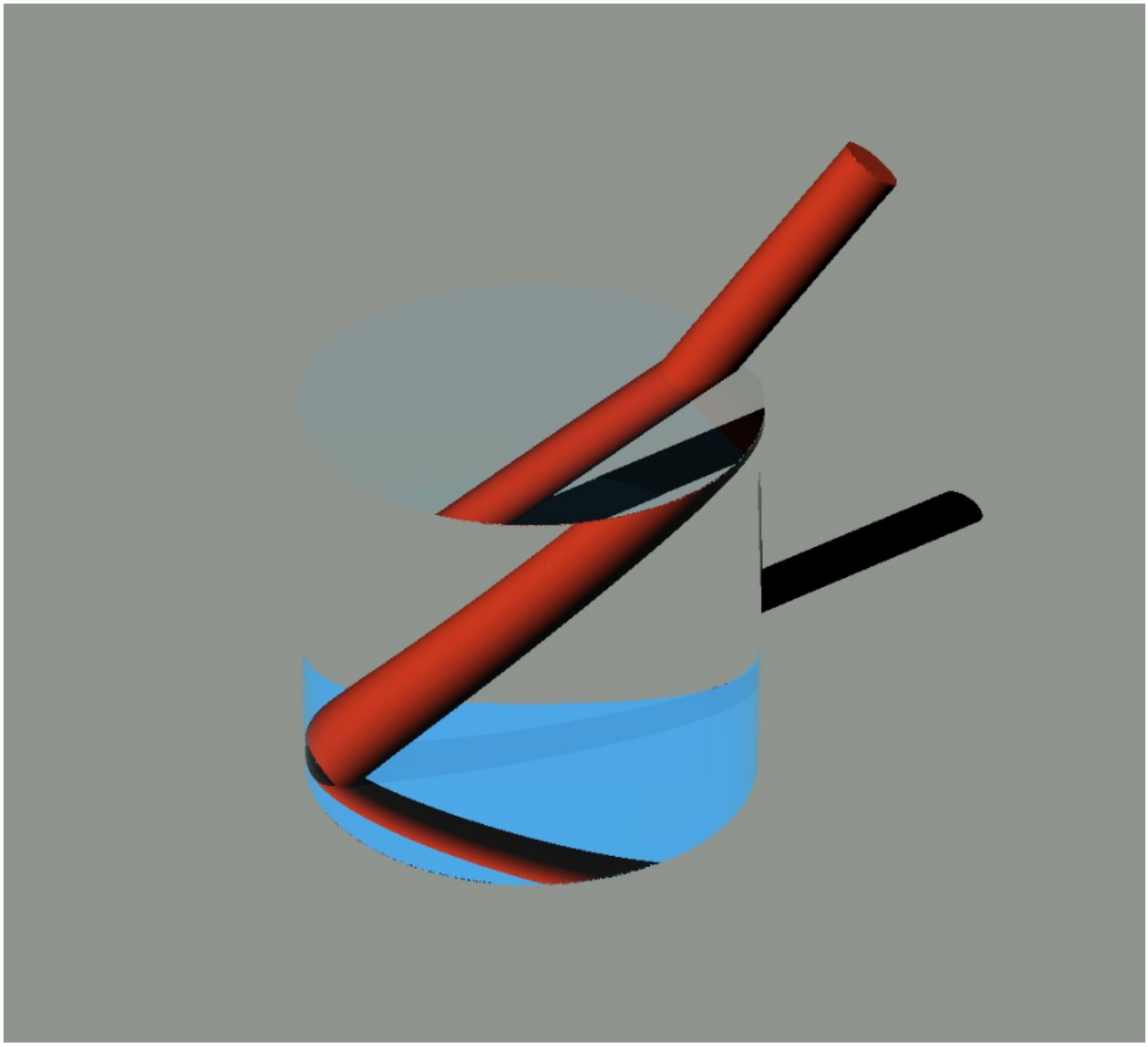
Il·lustració 3.6: Nou esferes amb propietats reflectives sobre un disc també reflectiu.



Il·lustració 3.7: Una esfera de vidre en la que s'hi pot veure el disc i l'esfera del redere.



Il·lustració 3.8: Una esfera i l'ombra de múltiples fonts de llum esfèriques.



Il·lustració 3.9: Efecte de la canyeta trencada dins d'un vas d'aigua.

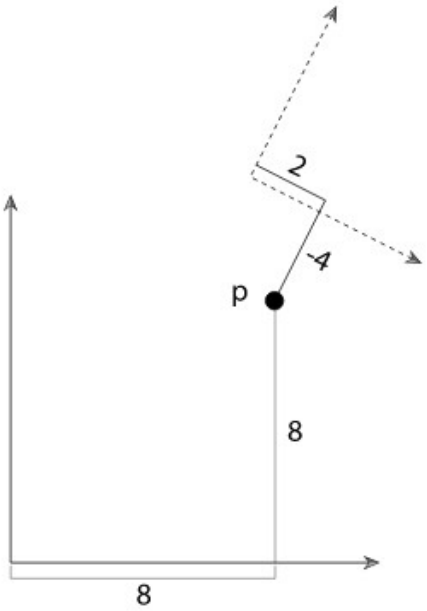
4. GEOMETRIA

En aquest apartat es presenten les bases geomètriques i matemàtiques sobre les que s'ha construït aquest algoritme de ray tracing. Els punts, vectors i matrius són essencials per al desenvolupament d'un programa d'aquest estil i en aquest punt veurem la seva implementació i les tècniques usades per manipular-los.

El codi presentat en aquest apartat es pot trobar en el mòdul `matrix`.

4.1. Sistema de coordenades

Com és típic en la computació gràfica, aquest ray tracer representa punts, vectors i normals en tres valors de coordenades: *x, y* i *z*. Aquests valors no tenen cap mena de sentit sense un sistema de coordenades que defineix l'origen de l'espai i proporciona tres vectors diferents que defineixen els eixos *x, y* i *z*. L'origen i els tres vectors formen un marc de referència que defineix l'eix de coordenades. Donat un determinat punt en 3D, les seves coordenades (*x, y*, z) depenen de la relació amb el marc. La Il·lustració 4.1 ens dona un exemple d'aquesta idea en 2D.



Il·lustració 4.1: En 2D, les coordenades (x, y) del punt «p» estan definides per la relació que té amb un particular sistema de coordenades. En aquesta imatge podem veure dos sistemes de coordenades; el punt pot tenir les coordenades (8, 8) respecte el sistema que té els eixos pintats en línies sòlides, però té les coordenades (2, -4) respecte el sistema que té els eixos pintats en línies discontinues. En qualsevol dels casos, el punt en 2D «p» està en la mateixa posició absoluta de l'espai.

En un cas de *n* dimensions, l'origen del marc de referència p_0 i els *n* vectors base independents defineixen un espai afí de *n* dimensions. Tots els vectors *v* en l'espai poden ser expressats com una combinació lineal dels vectors base. Donat un vector *v* i els vectors base v_i, hi ha un conjunt de valors escalars s_i de tal manera que

$$v = s_1 v_1 + ... + s_n v_n .$$

Els escalars s_i son representacions de *v* respecte els base { v1, v2, ... ,vn } i són els valors de la coordenada que guardem en el vector. De la mateixa manera, per tots els punts *p*, hi ha valors escalars únics s_i de tal manera que pot ser representat mitjançant l'origen p_o i els vectors base

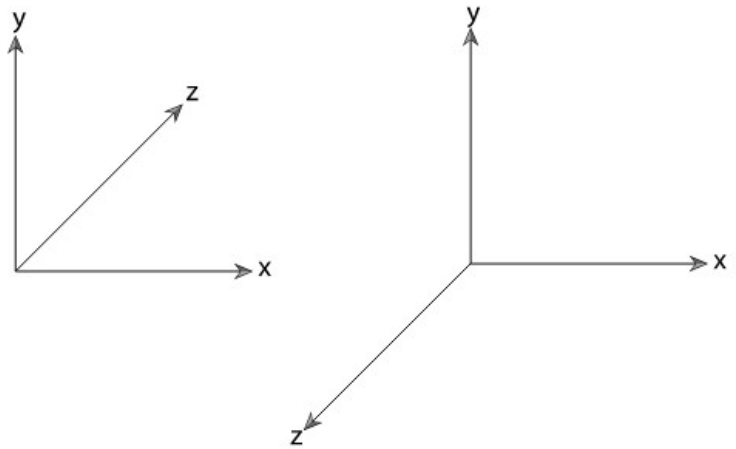
$$p = p_0 + s_1 v_1 + ... + s_n v_n$$

De tota manera, encara que tant vectors com punts estan representats amb coordenades *x, y* i *z*, són diferents entitats matemàtiques i no es poden intercanviar lliurement.

Aquesta definició de punts i vectors en un eix de coordenades presenta una paradoxa: per definir el marc de referència necessitem un punt i un conjunt de vectors, però només podem parlar significativament de punts i vectors quan van relacionats amb un sistema de coordenades. Per tant, en un sistema de tres dimensions, necessitem un *marc estàndard* amb l'origen a (0, 0, 0) i els vectors base (1, 0, 0), (0, 1, 0), i (0, 0, 1). La resta de marcs de referència estaran definits a partir d'aquest, el qual anomenem «*world space*».

4.1.1. Orientació del sistema de coordenades

Com es mostra en la il·lustració 6, hi ha dues maneres d'orientar el sistema de coordenades. Donats dos eixos *x* i *y* perpendiculars entre ells, l'eix *z* pot anar orientat en dos direccions diferents. Aquestes dos opcions s'anomenen sistema de la mà dreta i sistema de la mà esquerra. L'elecció sobre quin sistema es fa servir és arbitrari. En aquest projecte es fa ús del sistema de la mà dreta.



Il·lustració 4.2: (esquerra) En un sistema de la mà esquerra, l'eix z apunta cap a dintre de la pàgina quan l'eix x està orientat cap a la dreta i l'eix y cap amunt. (dreta) En un sistema de la mà dreta, l'eix z apunta cap a fora de la pàgina.

4.2. Punts i vectors

Aquest ray tracer permet treballar amb punts i vectors de dos i tres dimensions. S'ha creat un `struct` per punts i vectors en 2D i un altre pels de 3D. Encara que es faci servir el mateix `struct` per representar un vector i un punt, s'ha de tenir clar que no es tracta de la mateixa entitat, i per tant, no es poden aplicar les mateix operacions. Per exemple: es pot aplicar una translació sobre un punt, però per un vector no té sentit.

```
typedef struct {
  float x, y, z;
} Vec3f;

typedef struct {
  float x, y;
} Vec2f;
```

A continuació, ens centrarem en les operacions de vectors i punts en tres dimensions ja que són els més utilitzats en aquest projecte.

La implementació de punts i vectors s'ha fet mitjançant un `struct` de `floats`, els quals representen les coordenades *x, y* i *z,* corresponentment. S'ha seguit aquest model per facilitar la manera de treballar-hi, d'aquesta manera podem fer funcions que ens retornin directament un `struct` amb els valors desitjats i també podem donar aquest `struct` com a argument en les funcions.

S'ha creat la funció `makeVec3f()` per facilitar la creació d'aquests `structs`

```
Vec3f makeVec3f(float x, float y, float z) {
  Vec3f v = {.x=x, .y=y, .z=z};
  return v;
}
```

4.2.1. Mòdul d'un vector

El mòdul o llargada d'un vector ve determinat per la següent formula:

$$||V|| = \sqrt{V.x * V.x + V.y * V.y + V.z * V.z}$$

La notació pel mòdul d'un vector són aquestes dos barres a banda i banda, sent $||V||$ la llargada o mòdul del vector V .

La seva implementació en C:

```
float modVec3f(Vec3f a) {
  return sqrtf(a.x*a.x + a.y*a.y + a.z*a.z);
}
```

4.2.2. Normalització

Sovint és necessari normalitzar un vector. Això és calcular un nou vector que apunta a la mateixa direcció però amb una llargada igual a 1. Un vector normalitzat també se sol anomenar vector

unitari. La notació per un vector normalitzat és $\hat{v}$ la versió normalitzada de v. Per normalitzar un vector es útil saber-ne el mòdul.

`normalizeVec3f()` normalitza un vector. Divideix cada component per la llargada del vector, $\|v\|$. Retorna un nou vector; no modifica el vector que se li passa com a argument.

```
Vec3f normalizeVec3f(Vec3f a) {
  return multVec3f(a, 1/modVec3f(a));
}
```

`multVec3f()` multiplica el vector (primer argument) per un escalar (segon argument) i retorna un nou vector; no modifica el vector passat com a argument.

4.2.3. Producte escalar

El producte escalar és una operació molt comuna en qualsevol aplicació 3D. Es pot entendre com la projecció d'un vector sobre un altre i té com a resultat un escalar. Per dos vectors v i w, el seu producte escalar $(v \cdot w)$ està definit com:

$$v_x w_x + v_y w_y + v_z w_z$$

```
float dotProduct(Vec3f a, Vec3f b) {
  return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

El producte escalar té relació amb l'angle que hi ha entre els dos vectors:

$$(v \cdot w) = \|v\| \, \|w\| \cos\theta$$

on θ és l'angle entre v i w, i $\|v\|$ denota la longitud del vector v. D'aquí surt que el producte escalar serà zero si i només si v i w son perpendiculars sempre i quan tant v com w siguin diferent a $(0,0)$. Un conjunt de dos o més vectors perpendiculars entre ells es diu que són ortogonals. Un conjunt de vectors ortogonals unitaris s'anomena ortonormal.

De l'equació del producte escalar també en podem treure que si v i w són vectors unitaris, el producte escalar serà el cosinus de l'angle entre ells.

4.2.4. Producte vectorial

El producte vectorial és una altre operació molt útil per vectors de tres dimensions. A diferència del producte escalar, el producte vectorial retorna un vector. Donats dos vectors de tres dimensions, el producte vectorial $v \times w$ retorna un vector que és perpendicular als dos. Si v i w són dos vectors ortogonals, aleshores el seu producte escalar $v \times w$ forma un vector de tal manera que $(v, w, v \times w)$ formen un sistema de coordenades ortogonal.

El producte vectorial està definit com:

$$(v \times w)_x = v_y w_z - v_z w_y$$
$$(v \times w)_y = v_z w_x - v_x w_z$$
$$(v \times w)_z = v_x w_y - v_y w_x$$

```
Vec3f crossVec3f(Vec3f a, Vec3f b) {
  Vec3f v;
  v.x = (a.y * b.z) - (b.y * a.z);
  v.y = (a.z * b.x) - (b.z * a.x);
  v.z = (a.x * b.y) - (b.x * a.y);
  return v;
}
```

Una manera de recordar aquesta formula, és calcular el determinant se la següent matriu:

$$v \times w = \begin{bmatrix} i & j & k \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix}$$

on *i*, *j*, i *k* representen els eixos $(1,0,0)$, $(0,1,0)$, i $(0,0,1)$ respectivament. S'ha de tenir en compte que aquesta equació és només una ajuda de memorització i no una construcció matemàtica rigorosa, ja que barreja escalars i vectors.

De la definició del producte vectorial en podem treure que

$$||v \times w|| = ||v||\ ||w||\ |\sin\theta|\ ,$$

on θ és l'angle entre v i w. Una aplicació important d'això és que el producte vectorial de dos vectors perpendiculars unitaris també es un vector unitari. Si $v = (1,0,0)$ i $w = (0,1,0)$, aleshores $v \times w = (0,0,1)$. També es important veure que l'ordre dels vectors afecta al resultat de l'operació. Per exemple, agafant el cas anterior veiem que $w \times v$ no ens dona el mateix resultat que $v \times w$: $w \times v = (0,0,-1)$.

Diem doncs, que el producte vectorial és anti-commutatiu (intercanviar la posició dels dos arguments nega el resultat): si $v \times w = s$, aleshores $w \times v = -s$

4.2.5. Altres operacions

Hi ha altres operacions matemàtiques implementades però son més directes. La multiplicació d'un vector per un escalar o per un altre escalar dona un punt. Els vectors es poden sumar, restar, dividir entre ells, etc. Hi ha altres programes de renderització que fan distincions entre la implementació de punts, vectors i normals. Tècnicament presenten subtils diferències que podrien justificar la creació de tres entitats diferents. Per exemple: restar dos punts dona com a resultat un vector, sumar un vector a un altre vector o a un punt resulta en un punt, etc. De totes maneres, en aquest cas la complexitat que representava crear tres entitats diferents no sortia a compte en relació als avantatges que ens aportava. De la mateixa manera que OpenEXR, que ha esdevingut un estàndard en el món de la renderització, en aquest projecte es representen tots els tipus amb una sola entitat anomenada Vec3f. Així doncs, no es fan distincions entre normals, vectors i punts des d'un punt de vista de programació. Si que es tenen en compte però, els casos en que variables que representen diferents tipus de dades (normals, vectors, punts) però que estan declarades com a tipus genèric Vec3f, s'han de processar diferent. A continuació es presenta la implementació de les operacions més comunes:

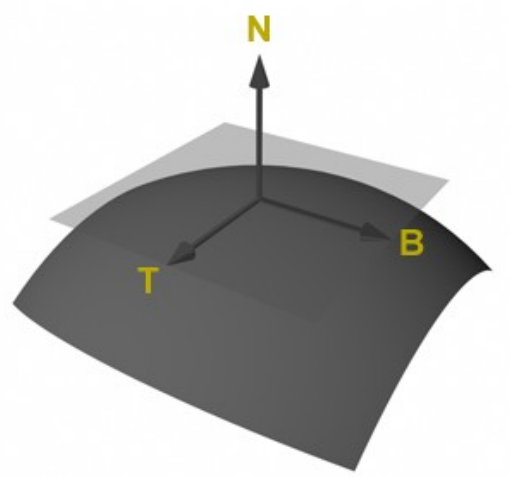
```
Vec3f addVec3f(Vec3f a, Vec3f b) {
  return makeVec3f(a.x + b.x, a.y + b.y, a.z + b.z);
}

Vec3f subVec3f(Vec3f a, Vec3f b) {
  return makeVec3f(a.x - b.x, a.y - b.y, a.z - b.z);
}

Vec3f multVec3f(Vec3f a, float b) {
  return makeVec3f(a.x * b, a.y * b, a.z * b);
}
Vec3f multByVec3f(Vec3f a, Vec3f b) {
  return makeVec3f(a.x * b.x, a.y * b.y, a.z * b.z);
}
```

4.3. Vector normal

El vector normal és un terme que es fa servir en el comput gràfic i en la geometria per descriure la orientació d'una superfície en un punt concret. Es pot descriure com el producte vectorial entre dos vectors no paral·lels que són tangents a la superfície en un punt específic. Amb altres paraules és un vector perpendicular a la superfície en un punt en concret.

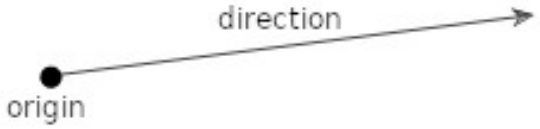


Il·lustració 4.3: La normal (N) es pot descriure com el producte vectorial de dos vectors no paral·lels tangents a la superfície (T i B) en un punt en concret.

El vector normal és molt important per calcular la brillantor amb la que veiem un objecte ja que juntament amb els raigs d'una font de llum determinen quanta llum arriba a la superfície de l'objecte. Això està explicat detalladament al punt 7.2.

4.4. Raigs

Un raig és una recta semi infinita definida per un origen i una direcció. En aquest projecte es representa un raig com un struct que conté dos membres. Aquests membres són del tipus `Vec3f` (explicant al punt 4.2) i representen l'origen i la direcció del raig; com ja hem dit anteriorment s'utilitza el mateix tipus per representar tant un punt com un vector.



Il·lustració 4.4: Un raig és una recta semi infinita definida pel seu origen (origin) i la seva direcció (direction).

La declaració de raig es troba en el mòdul `rays` i és la següent:

```
typedef struct {
  Vec3f origin;
  Vec3f direction;
} Ray;
```

El raig també té una funció implementada per facilitar-ne la creació:

```
Ray rays_make(Vec3f origin, Vec3f direction) {
  Ray ray = {.origin = origin, .direction = direction};
  return ray;
}
```

La forma paramètrica d'un raig s'expressa en funció d'un valor escalar t que dona el conjunt de tots els punts pels quals el raig passa:

$$r(t) = origin + t * direction \quad 0 \leqslant t \leqslant \infty$$

4.5. Matrius

En aquest projecte també s'ha declarat un tipus per treballar amb matrius així com unes operacions bàsiques per poder tractar-les. Les matrius ens permeten combinar totes les transformacions que es poden aplicar sobre punts o vectors (translació, rotació i escalat) en una sola operació. Per exemple; podem generar una matriu que roti un punt 90º sobre l'eix de les x, l'escali per 2 en l'eix de les z (l'escalat aplicat sobre el punt seria $(1,1,2)$) i li apliqui una translació per $(-2,3,1)$. Aleshores només hem de multiplicar la matriu pel punt en qüestió i obtenim el punt transformat. Això també es podria fer aplicant una sèrie de transformacions lineals sobre el punt però seria més llarg i implicaria escriure més codi per fer-ho.

Típicament, en el món de la computació gràfica es fa ús de matrius quadrades, que són matrius amb el mateix nombre de files que de columnes, i en aquest projecte ens limitarem a l'ús de les matrius 4x4.

Les matrius, en aquest projecte, s'han fer servir bàsicament a l'hora de traçar els raigs de la càmera cap a l'escena. Com veurem més endavant amb detall en l'apartat 5, donada una càmera amb una ubicació i una orientació concretes s'han de traçar raigs cap aquesta direcció, i la matriu ens va molt be per fer totes aquestes transformacions de manera més simple.

A continuació entrarem en detall de com i per què funcionen les matrius.

4.5.1. Multiplicació punt-matriu

Un punt o un vector són una seqüència de tres nombres i per tant, també poden ser expressats com a matrius 1×3, matrius que tenen una fila i tres columnes. I si tenim punts que els podem expressar com a matrius, també els podem multiplicar per altres matrius. Recordem que una matriu $m \times p$ es pot multiplicar per una matriu $p \times n$ per donar com a resultat una matriu $m \times n$. Si la primera matriu és un punt, aleshores podem dir que $m = 1$ i $p = 3$, el que implica que es podrà multiplicar per les matrius $p \times n$ que siguin de la forma $3 \times n$, on n pot ser qualsevol valor més gran o igual a 1. Exemple de multiplicació de matrius $[1 \times 3] * [3 \times 4]$:

$$[x \quad y \quad z] * \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \end{bmatrix}$$

Al multiplicar un punt per una matriu transforma el punt en una nova posició. El resultat d'un punt per una matriu ha de ser sempre un punt. Si representem un punt com una matriu $[1 \times 3]$, hem de multiplicar-lo per una matriu $[3 \times 3]$ perquè el resultat sigui una matriu $[1 \times 3]$ (un altre punt).

$$[x \quad y \quad z] * \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix}$$

El mon del comput gràfic, i com hem dit abans, ens centrarem en fer servir matrius 4×4, i aviat s'explicarà perquè, de moment però, ens centrarem en les matrius 3×3.

4.5.2. Matriu identitat

La matriu identitat és una matriu quadrada, la qual tots els seu quocients excepte els de la diagonal són 0. Els quocients de la diagonal són 1:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

El resultat d'un punt multiplicat per la matriu identitat és el mateix punt.

4.5.3. Matriu d'escalat

Quan es multiplica un punt p per una matriu, les coordenades d'aquest punt es multipliquen respectivament pels coeficients C_{00} per x, C_{11} per y i C_{22} per z. Quan aquests coeficients són 1 i tota la resta 0, tenim la matriu identitat. Per altre banda, si aquests coeficients són diferents de 0, actuen com a multiplicadors de les coordenades del punt, o en altres paraules, les coordenades s'escalen cap amunt o cap avall depenent dels valors. La matriu d'escalada es pot escriure com:

$$\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & Sz \end{bmatrix}$$

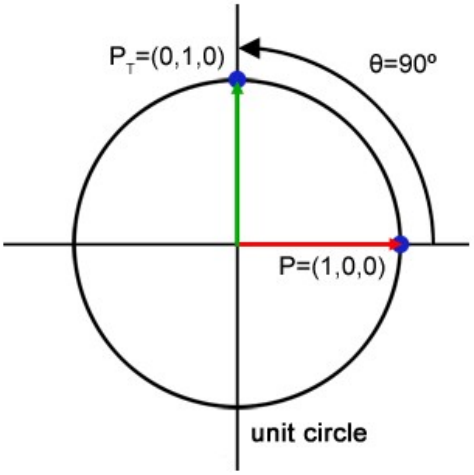
On els nombres reals Sx, Sy i Sz, són els factors d'escalat.

Si algun d'aquests coeficients de la matriu és negatiu, aleshores la coordenada del punt pel respectiu eix s'invertirà.

4.5.4. Matriu de rotació

En aquest punt s'explica com es crea una matriu per rotar un punt o un vector al voltant d'un eix del sistema de coordenades. Per fer-ho és necessari fer servir equacions trigonomètriques.

Agafem un punt P definit en un sistema de coordenades 3D $(1,0,0)$. Ignorem l'eix z per un moment i assumim que el punt està situat en en pla xy. El que s'ha de fer és transformar el punt P en el punt P_T mitjançant una rotació (també es podria fer amb una translació).



Il·lustració 4.5: 90 graus en el sentit contrari de les agulles del rellotge.

Assumim ara doncs, que tenim una matriu R. Quan multipliquem P per R, P es transforma en P_T. Considerant el que hem explicat fins, reescriurem la multiplicació de punt-matriu per intentar aïllar el càlcul de cada una de les coordenades transformades.

$$P_T.x = P.x * R_{00} + P.y * R_{10} + P.z * R_{20}$$
$$P_T.y = P.x * R_{01} + P.y * R_{11} + P.z * R_{21}$$
$$P_T.z = P.x * R_{02} + P.y * R_{12} + P.z * R_{22}$$

Com ja hem dit, ara per ara, $P_T.z$, que representa la coordenada z de P_T no ens interessa, Així que ens centrarem en $P_T.x$ i $P_T.y$:

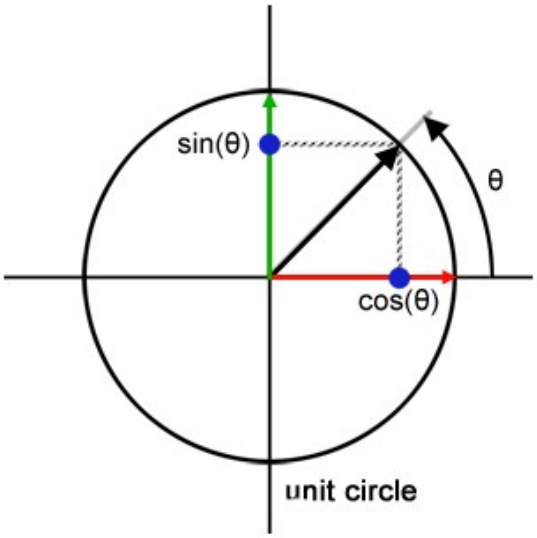
- De P a P_T, la coordenada x va de 1 a 0. Si mirem a la primera línia de les equacions que hem escrit per calcular P_T, això vol dir que R_{00} ha de ser 0. Considerant que $P.y$ i $P.z$ son 0, ja no ens preocupem del valor de R_{10} i R_{20}.
- De P a P_T, la coordenada y va de 0 a 1. Si mirem a la segona línia de les equacions escrites per calcular P_T, veiem que l'única opció que tenim és posar el valor $R_{01} = 1$. Ja que $P = (1,0,0)$.

Si agafem els valors que tenim i els escrivim en una matriu, obtenim:

$$R_z = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Per tant, $P * R_z = P_T$.

A partir d'aquí es comencen a fer servir les funcions trigonomètriques. Si mirem al punt en el cercle unitari veiem que podem calcular les coordenades x i z fent servir el sinus i el cosinus de l'angle θ (il·lustració 4.6).



Il·lustració 4.6: Podem fer servir el sinus i el cosinus per calcular les coordenades d'un punt en els eixos x i y d'un cercle unitari.

$$x = \cos(\theta) = 0$$
$$y = \sin(\theta) = 1$$
$$\text{with } \theta = \tfrac{\pi}{2}$$

De manera que podríem reescriure la matriu R_z com:

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ with } \theta = \frac{\pi}{2}$$

Si en comptes de rotar el punt 90 graus, només el volem rotar 45, substituïm l'angle per 45º o $\pi/4$, i apliquem R_z a P, obtenim les coordenades $(0.7071, 0.7071)$ per P_T, que és correcte. Per tant, sembla que podem generalitzar una matriu que apliqui rotacions al voltant de l'eix z:

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Ara imaginem que tenim un punt $P=(0,1,0)$ i $P_T=(1,0,0)$, que en aquest cas és aplicar una rotació de 90º però en el sentit de les agulles del rellotge (il·lustració 4.7).

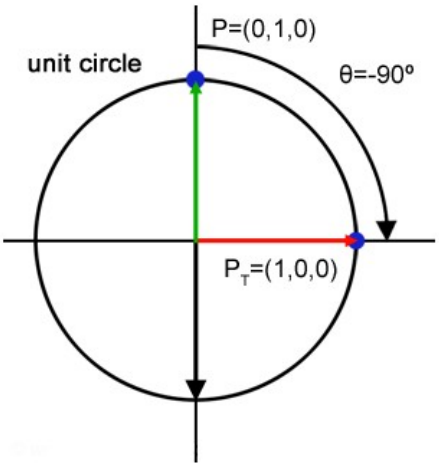
$$R_z = \begin{bmatrix} \cos(-\frac{\pi}{2}) & \sin(-\frac{\pi}{2}) & 0 \\ \sin(-\frac{\pi}{2}) & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} P_T.x = \; & 0*R_{00} + 1*R_{10} + P.z*R_{20} = \\ & 0*0 + 1*-1 + 0*0 = -1 \\ P_T.y = \; & 0*R_{01} + 1*R_{11} + P.z*R_{21} = \\ & 0*-1 + 1*0 + 0*0 = 0 \\ P_T.z = \; & 0*R_{02} + 1*R_{12} + P.z*R_{22} = \\ & 0*0 + 1*0 + 0*1 = 0 \end{aligned}$$

Com es veu, si fem servir aquesta matriu per aplicar rotacions en el sentit de les agulles del rellotge no dona un resultat correcte, ja que hem obtingut les coordenades $(-1,0,0)$ en comptes de $(1,0,0)$. Per obtenir les coordenades correctes, R_{10} hauria de ser 1 (i no -1). En aquest cas, la matriu R seria:

$$R_z = \begin{bmatrix} \cos(-\frac{\pi}{2}) & \sin(-\frac{\pi}{2}) & 0 \\ -\sin(-\frac{\pi}{2}) & \cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} P_T.x = \; & 0*R_{00} + 1*R_{10} + P.z*R_{20} = \\ & 0*0 + 1*1 + 0*0 = 1 \\ P_T.y = \; & 0*R_{01} + 1*R_{11} + P.z*R_{21} = \\ & 0*-1 + 1*0 + 0*0 = 0 \\ P_T.z = \; & 0*R_{02} + 1*R_{12} + P.z*R_{22} = \\ & 0*0 + 1*0 + 0*1 = 0 \end{aligned}$$



Il·lustració 4.7: Rotació de 40º en el sentit de les agulles del rellotge.

Sabem que els punts en un pla xy s'haurien de quedar en aquest pla si els rotem al voltant de l'eix z (de manera que la matriu Rz no hauria d'afectar la coordenada z de P_T). Quan mirem a les equacions per transformar P en P_T, es fàcil veure que la tercera fila i la tercera columna no afecten al càlcul de P_T, ja que els primers dos coeficients per calcular el valor de $P_T.z$; R_{02} i R_{12} són 0 i el tercer, R_{22}, és 1, que multiplicat pe r P_z deixa el seu valor igual. Per tant, podem concloure que la matriu per rotar un punt o vector al voltant de l'eix z és la següent:

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Per trobar les matrius per aplicar rotació en els eixos x i y, es poden seguir els mateixos passos que s'han fet per arribar a la matriu R_z:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

4.5.5. Combinació de matrius

Com ja hem comentat, es pot fer servir una sola matriu per aplicar més d'una transformació en un punt o un vector. Per fer aquesta combinació només s'han de multiplicar les dos matrius que volem combinar. Posem per cas que tenim una matriu que aplica un escalat (E), una altre que aplica una rotació en l'eix x (Rx), i una última que aplica una rotació en l'eix y (Ry). El que hem de fer és multiplicar-les per acabar amb una sola matriu que apliqui les tres transformacions:

$$ER_x = E * R_x$$
$$ER_{xy} = ER_x * R_y$$

S'ha de tenir en compte que l'ordre en que multipliquem les matrius té importància. En aquest cas, la matriu resultant seria equivalent a primer escalar el punt, després aplicar la rotació al voltant de l'eix x i finalment aplicar la rotació al voltant de l'eix y.

4.5.6. Relació entre matrius i sistemes de coordenades

Imaginem que tenim un punt P_x amb les coordenades (1,0,0) i el volem rotar al voltant de l'eix z 10 graus en el sentit de les agulles del rellotge. Aplicant el que hem vist anteriorment, sabem que és fàcilment calculable amb equacions trigonomètriques. La coordenada x, la trobem fent cos(-10) i la y fent sin(-10). Si repetim el procés però ara amb un punt P_x amb les coordenades (0,1,0), aleshores la coordenada x serà -sin(-10) i la coordenada y, cos(-10). Es pot observar que la primera fila de la matriu de rotació al voltant de l'eix z conté les mateixes funcions que s'han fet servir per calcular les noves coordenades del punt P_x. També podem veure que la segona línia d'aquesta mateixa matriu conté les mateixes funcions que les que s'han fet servir per calcular les noves coordenades de P_y:

$$Px_x = \cos(\theta) \quad Px_y = \sin(\theta)$$
$$Py_x = -\sin(\theta) \quad Py_y = \cos(\theta)$$

La idea fonamental de les matrius és que cada fila representa un dels eixos d'un sistema de coordenades. L'orientació (rotació), mida (escalat) i posició (translació) d'aquest sistema de coordenades representa la transformació que s'aplicarà als punts quan siguin multiplicats per aquesta matriu. Els punts, originalment són definits en un sistema de coordenades el qual anomenarem A. Si un punt està lligat a un altre sistema de coordenades B (la matriu), que movem i rotem, les coordenades del punt no canviaran respecte B. El punt està d'alguna manera lligat a les transformacions aplicades en el sistema B (és mou juntament amb el sistema de coordenades). De totes maneres, les coordenades del punt si que canvien en el sistema de coordenades A. Multiplicant un punt que té les coordenades expressades en A per la matriu B, ens donarà les noves coordenades del punt en A.

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} \begin{matrix} \rightarrow \\ \rightarrow \\ \rightarrow \end{matrix} \begin{matrix} x-axis \\ y-axis \\ z-axis \end{matrix}$$

4.6. Transformacions

Les transformacions consisteixen en canviar un punt o vector de lloc. El que hem estat veient en l'apartat anterior de matrius eren els principis bàsics per crear les matrius adequades per poder aplicar transformacions en punts i vectors. Ara veurem com aplicar aquestes matrius i portar a terme les transformacions.

Fins ara havíem estat tractant amb matrius 3x3 per facilitar les coses, però com ja s'ha comentat abans en el còmput gràfic es fan servir les matrius 4x4 i a continuació veurem per què.

4.6.1. Transformacions sobre punts

S'ha parlat de diferents tipus de transformacions que es poden aplicar sobre punts, de totes maneres, sobre la translació, que és una de les operacions lineals més simples que es poden aplicar en un punt no se n'ha parlat massa. Per aconseguir aplicar una translació amb la teoria de matrius que hem explicat s'han de realitzar certs canvis.

Com ja s'ha mencionat, només es pot portar a terme una multiplicació de matrius si tenen unes dimensions compatibles: $m \times p$ i $p \times n$. Se sap que un punt multiplicat per la matriu identitat deixa les seves coordenades intactes. Aplicar una translació en un punt és sumar certs valors a les seves coordenades. Es posa per cas que es vol moure un punt (1,1,1), a la coordenada (2,3,4). Per fer-ho s'han de sumar els valors 1, 2 i 3 a les coordenades x, y i z, en el punt original.

$$P'.x = P.x * M_{00} + P.y * M_{10} + P.z * M_{20} + T_X$$
$$P'.y = P.x * M_{01} + P.y * M_{11} + P.z * M_{21} + T_Y$$
$$P'.z = P.x * M_{02} + P.y * M_{12} + P.z * M_{22} + T_Z$$

T és el valor que s'ha de sumar a cada coordenada per aplicar la translació. Si es volgués generar una matriu amb la que també es poguessin fer translacions, s'haurien d'integrar Tx, Ty i Tz a la matriu. Per calcular la component x del punt transformat, per exemple, només es fan servir els coeficients de la matriu de la primera columna. Si la matriu tingues quatre coeficients en comptes de 3, Tx passaria a ser M_{30}:

$$P'.x = P.x * M_{00} + P.y * M_{10} + P.z * M_{20} + M_{30}$$
$$P'.y = P.x * M_{01} + P.y * M_{11} + P.z * M_{21} + M_{31}$$
$$P'.z = P.x * M_{02} + P.y * M_{12} + P.z * M_{22} + M_{32}$$

Al fer això però, la matriu passa a ser 4x3, de manera que ja no es poden multiplicar els punts 1x3 per aquestes noves matrius. Com a solució es pot afegir un altre columna addicional al punt i que el seu valor sempre sigui 1. La representació dels punts ara té aquesta forma doncs: (x,y,z,1). En el món de la computació gràfica això s'anomena un punt homogeni(o un punt amb coordenades homogènies).

Per tant, per poder treballar amb matrius que puguin aplicar translacions, rotacions i escalats, s'ha de treballar amb punts que tinguin coordenades homogènies. Com que el quart valor sempre és un 1, en el codi s'obvia. Es defineixen x, y i z i s'assumeix que hi ha un quart valor.

La matriu aconseguida fins ara te la forma 4x3, que no es la 4x4 que s'ha dit que normalment es fa servir en el món de la computació gràfica. La quarta columna de la matriu serveix per implementar altres tipus de transformacions que no són gaire comunes i normalment es deixa amb els valors (0,0,0,1).

La implementació de la funció per aplicar les transformacions d'una matriu sobre un punt queda de la següent manera:

```
Vec3f multVecMatrix(Vec3f src1, Matrix44f src2) {
  Vec3f v;
  v.x = src1.x * src2[0][0] + src1.y * src2[1][0] + src1.z * src2[2][0] +
src2[3][0];
  v.y = src1.x * src2[0][1] + src1.y * src2[1][1] + src1.z * src2[2][1] +
src2[3][1];
  v.z = src1.x * src2[0][2] + src1.y * src2[1][2] + src1.z * src2[2][2] +
src2[3][2];
  return v;
}
```

4.6.2. Transformacions sobre vectors

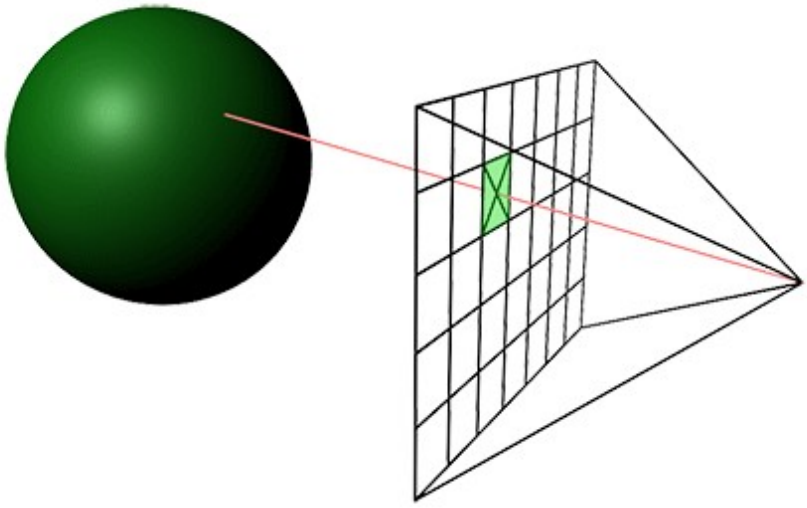
Les transformacions sobre vectors són més simples que les transformacions sobre punts. Els vectors, al representar direccions i no punts en l'espai no tenen una posició definida, així que no se'ls hi poden aplicar translacions.

La funció per aplicar les matrius de transformació doncs, es com la que s'ha implementat pels punts però sense considerar la part que està destinada a aplicar les translacions.

```
Vec3f multDirMatrix(Vec3f src1, Matrix44f src2) {
  Vec3f dst;
  dst.x = src1.x * src2[0][0] + src1.y * src2[1][0] + src1.z * src2[2][0];
  dst.y = src1.x * src2[0][1] + src1.y * src2[1][1] + src1.z * src2[2][1];
  dst.z = src1.x * src2[0][2] + src1.y * src2[1][2] + src1.z * src2[2][2];
  return dst; }
```

5. GENERACIÓ DE RAIGS PRIMARIS

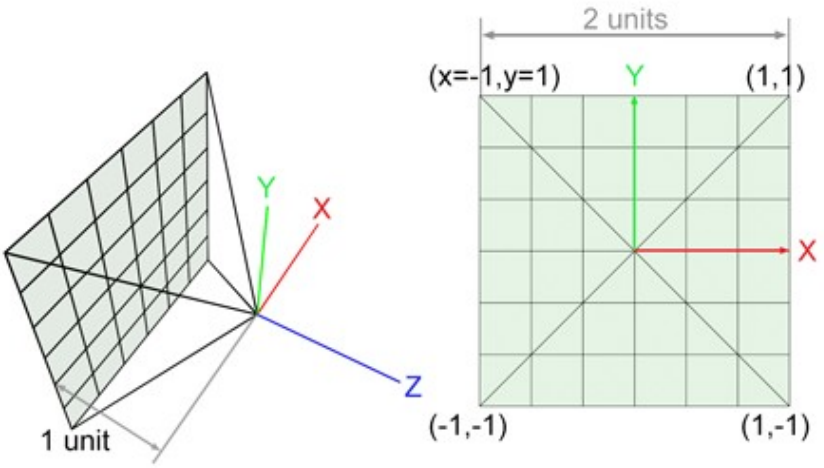
Com ja hem comentat, la tasca d'un renderitzador és assignar un color a cada pixel d'una imatge. El resultat ha de ser una representació precisa de l'escena que es vol representar des d'un punt de vista determinat. Com també hem comentat abans en el punt 3, per crear imatges amb un ray tracer hem de generar raigs per cada píxel de la imatge. Quan un raig col·lisiona amb un objecte de l'escena canvia el color del pixel pel que ha passat pel color de l'objecte amb el que ha col·lisionat (a l'apartat 7 d'ombreig veurem que és una mica més complicat). Aquest procés que ja l'hem comentat en el punt 3 s'anomena «backward-tracing».



Il·lustració 5.1: El «backward-tracing» consisteix en traçar raigs de la càmera cap a l'escena. Si el raig col·lisiona amb algun objecte el color del pixel pel que està passant el raig pren el color de l'objecte en el punt d'intersecció.

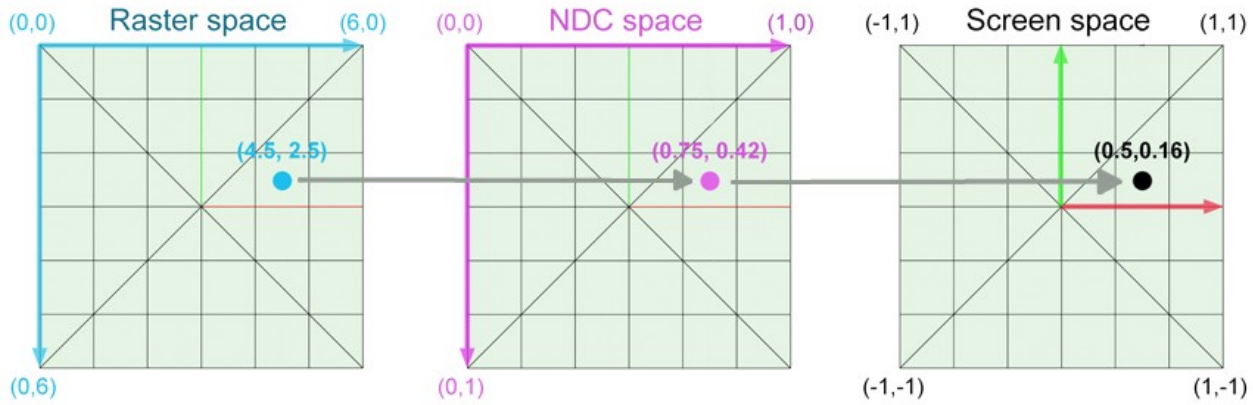
El procés de crear una imatge comença per construir aquests raigs anomenats raigs primaris (primaris perquè són els primers que «disparem» cap a l'escena. Els raigs secundaris poden ser els raigs d'ombreig, dels quals en parlarem en el punt 7). Aquests raigs comencen a l'origen de la càmera, i aquest origen, per convenció amb altres ray tracers és $(0,0,0)$ (l'origen del sistema de coordenades). Com hem vist abans en el punt 3, el film de la càmera estenopeica està situat darrere l'obertura de la càmera, que provoca que els raigs de llum formin una imatge invertida de l'escena. Aquesta inversió però, es pot evitar si situem el film en el mateix costat que l'escena (davant de l'obertura). Per convenció amb altres ray tracers, aquest film se situa a una unitat de l'origen de la càmera, i també per convenció, s'orienta la càmera cap a la part negativa de l'eix z. Finalment, per simplificar les coses, ara per ara suposarem que la nostre imatge renderitzada és quadrada (l'altura i l'amplada de la imatge en píxels és la mateixa).

Per traçar un raig per cada píxel de la imatge, hem de traçar una línia que vagi de l'origen de la càmera cap al centre de cada píxel (il·lustració 5.1). Es pot expressar aquesta línia com un raig l'origen del qual és l'origen de la càmera i la seva direcció és el vector que va de l'origen de la càmera cap al centre del píxel. Per calcular la posició del centre d'un píxel, hem de convertir les coordenades del pixel, que originalment estan expressades en «raster space» (les coordenades estan expressades en píxels, sent $(0,0)$ la coordenada del píxel situat a la cantonada superior esquerra), a coordenades «world space».



Il·lustració 5.2: (esquerra) Una càmera bàsica. La mida original de la imatge és 6x6 píxels i l'origen de la càmera està situat a les coordenades $(0,0,0)$*. La càmera està orientada cap a la part negativa de l'eix* z *i el pla de la imatge està situat a una unitat de l'origen de la càmera. (dreta) Hi ha píxels que tenen coordenades negatives.*

«World space» és l'espai en que estan situats tots els objectes, llums i càmeres en l'escena. Per exemple, si una esfera està situada a 5 unitats de l'origen de coordenades en direcció a la part negativa de l'eix z, les seves coordenades en «world space» són $(0,0,-5)$. Si volem ser capaços de calcular matemàticament si es produeix una intersecció entre un raig i un objecte, l'origen i la direcció del raig han d'estar definits en el mateix sistema de coordenades que l'esfera (que es trobava en la posició $(0,0,-5)$).



Il·lustració 5.3: Convertir les coordenades d'un punt en el mig del píxel a «world space» requereix uns quants passos. Primer estan expressades en «raster space» (les coordenades del píxel amb un afegit de 0.5), després s'han de convertir a «NDC space» (traduïm les coordenades en un rang de [0,1]) i finalment les convertim a «screen space» (on tenen un rang de [-1,1]).

Per trobar la relació entre les coordenades d'un píxel en «raster space» i les en «world space», primer hem de normalitzar la posició d'aquest pixel fent servir la mida de la imatge. Les noves coordenades normalitzades, es diu que estan definides en «NDC space» (Normalized Device Coordinates):

$$PixelNDC_x = \frac{(Píxel_x + 0.5)}{ImageWidth},$$

$$PixelNDC_y = \frac{(Píxel_y + 0.5)}{ImageHeight}.$$

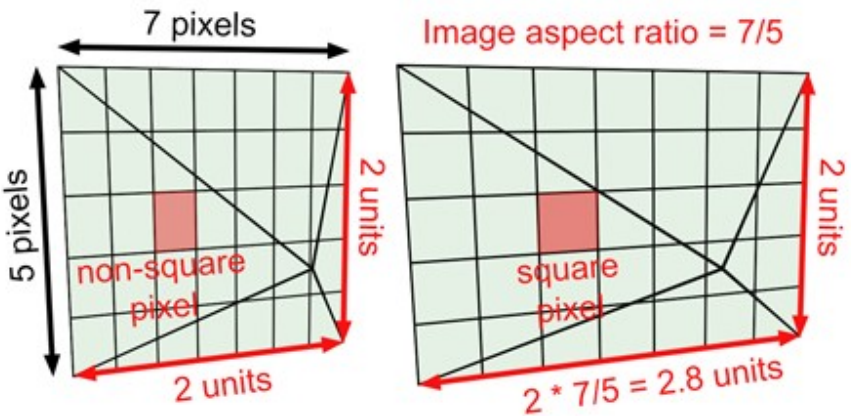
Sent $ImageWidth$ l'amplada de la imatge i $ImageHeight$ l'altura de la imatge. Afegim, tant horitzontalment com verticalment 0.5 a la posició del píxel perquè el raig que disparem des de la càmera passi pel centre del píxel. Les coordenades expressades en «NDC space» tenen un rang que va de [0,1]. Com podem veure en la il·lustració 5.2, la pel·lícula (o pla de la imatge) està centrada a l'origen del sistema de coordenades, per tant, els píxels ubicats a l'esquerra de la imatge haurien de tenir coordenades negatives en l'eix de les x i els píxels de la dreta haurien de tenir-les positives. Això també s'aplica a l'eix de les y: els píxels situats a dalt de la línia formada per l'eix x haurien de tenir coordenades y positives i els situats a sota les haurien de tenir negatives. Això se soluciona passant les noves coordenades normalitzades, que tenen el rang [0:1], a un altre rang que vagi de [-1:1]:

$$PixelScreen_x = 2 * PixelNDC_x - 1,$$
$$PixelScreen_y = 1 - 2 * PixelNDC_y.$$

Les coordenades definides d'aquesta manera se sol dir que estan en «screen space».

5.1. Relació d'aspecte

Fins ara, per simplificar les coses, hem assumit que la imatge era quadrada, però anem a fer possible que es puguin generar imatges amb una relació d'aspecte diferent. Imaginem que tenim una imatge que fa 7 per 5 píxels (és una imatge molt petita però no deixa de ser una imatge). Dividint l'amplada per l'altura de la imatge ens dona 1.4. Quan les coordenades del píxel estan definides en «screen space» estan en el rang [-1,1], però hi ha més píxels en l'eix de les x (7), que en l'eix de les y (5), per tant els píxels queden estirats i aixafats (il·lustració 5.4).



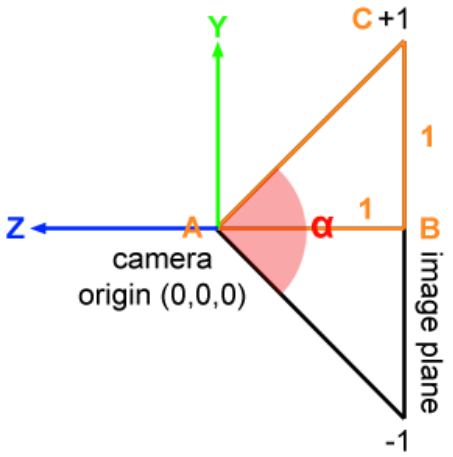
Il·lustració 5.4: (esquerra) Com que l'alçada i l'amplada de la imatge són diferents els píxels no són quadrats. Per corregir-ho s'ha d'escalar l'eix x de la imatge fent servir la relació d'aspecte.

Per tornar a fer els píxels quadrats, s'han de multiplicar les coordenades x dels píxels per la relació d'aspecte de la imatge, que en aquest cas és 1.4. Aquesta operació deixa les coordenades y dels píxels com estaven (rang [-1,1]), però les coordenades x dels píxels ara es troben en el rang [-1.4,1.4].

$$RelacióAspecte = \frac{ImageWidth}{ImageHeight},$$
$$PixelScreen_x = (2 * PixelNDC_x - 1) * RelacióAspecte,$$
$$PixelScreen_y = (1 - 2 * PixelNDC_y).$$

5.2. Camp de visió

Finalment només ens falta tenir en compte el camp de visió. Les coordenades y de qualsevol punt definides en «screen space» estan en el rang [-1,1]. També sabem que el pla de la imatge està situat a una unitat de l'origen de la càmera. Si mirem a aquesta configuració de la càmera des del costat, es pot dibuixar un triangle format per l'origen de la càmera, la part més alta del pla de la imatge i la part més baixa.



Il·lustració 5.5: Vista lateral de la configuració de la càmera. La distància de l'origen de la càmera (camera origin) al pla de la imatge (image plane) és una unitat (vector AB). La distància de B a C també és una unitat. Amb geometria simple podem calcular l'angle α.

Com que sabem la distància que hi ha entre l'origen de la càmera i el pla de la imatge (1 unitat) i l'alçada del pla de la imatge (2 unitats ja que va de y=1 a y=-1) podem calcular l'angle del triangle rectangle ABC que és la meitat de l'angle α, del qual n'estem interessats.

$$\frac{\alpha}{2} = atan\left(\frac{Costat\ oposat}{Costat\ adjecent}\right) = atan\left(\frac{1}{1}\right) = \frac{\pi}{4}.$$

En altres paraules, l'angle de visió, en aquest cas en particular és de 90 graus. Per calcular la llargada de la línia BC, el que hem de fer és calcular la tangent de l'angle α dividit per dos:

$$\|BC\| = \tan\left(\frac{\alpha}{2}\right).$$

Podem observar, per tant, que per angles majors de 90º, $\|BC\|$ serà més gran que 1, i que per angles menors de 90º, $\|BC\|$ serà més petit que 1. Per tant, podem multiplicar les coordenades de cada píxel per escalar-les amunt o avall. Aquesta operació canvia la quantitat d'escena que veiem d'una manera similar a com ho fa el zoom. En resum: podem definir el camp de visió de la càmera com l'angle α i multiplicar les coordenades dels píxels amb el resultat de la tangent d'aquest angle dividit per dos.

$$PixelScreen_x = (2 * PixelNDC_x - 1) * RelacióAspecte * \tan(\frac{\alpha}{2}),$$
$$PixelScreen_y = (1 - 2 * PixelNDC_y) * \tan(\frac{\alpha}{2}).$$

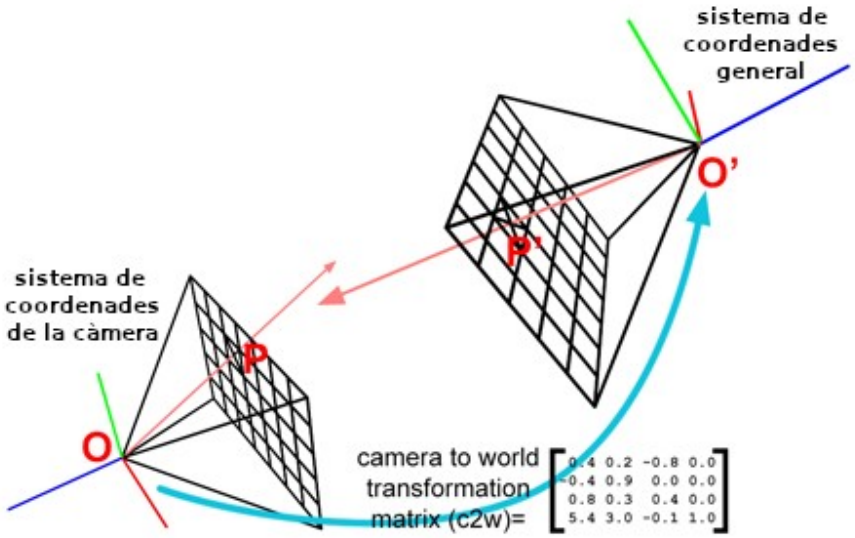
Ara les coordenades originals de cada píxel estan expressades en funció del pla de la imatge de la càmera, i en aquest punt es diu que estan en «camera space». Quan la càmera està en la seva posició per defecte, el sistema de coordenades de la càmera i el sistema de coordenades general (on es representen la resta d'objectes, llums...) estan alineats. Per tant, podem expressar la coordenada final d'un píxel com:

$$P_{cameraSpace} = (PixelCamera_x, PixelCamera_y, -1)$$

Això ens dona la posició P ($P_{cameraSpace}$) d'un píxel de la imatge en pla de la imatge de la càmera. A partir d'aquí podem calcular un raig per aquest píxel definint l'origen del raig com l'origen de la càmera (anomenem aquest punt com a O i la direcció del raig com el vector normalitzat OP . El vector OP és simplement la posició del píxel en el pla de la imatge menys l'origen de la càmera. L'origen de la càmera coincideix amb l'origen del sistema de coordenades general quan la càmera està en en la seva posició per defecte, de manera que O és simplement $(0,0,0)$.

5.3. Transformacions en la càmera

Finalment, hem de ser capaços de renderitzar una imatge des de qualsevol punt de vista. Després de moure la càmera de la seva posició original (centrada a l'origen del sistema de coordenades general i alineada amb la part negativa de l'eix z), es poden expressar les translacions i rotacions que apliquem a la càmera en una matriu 4x4. Normalment aquesta matriu s'anomena matriu «camera-to-world». Si apliquem aquesta matriu als punts O i P, aleshores el vector $\|O'P'\|$ (on O' és el punt O i P' el punt P transformats per la matriu «camera-to-world») representa la direcció normalitzada del raig en el sistema de coordenades general (il·lustració 5.6).

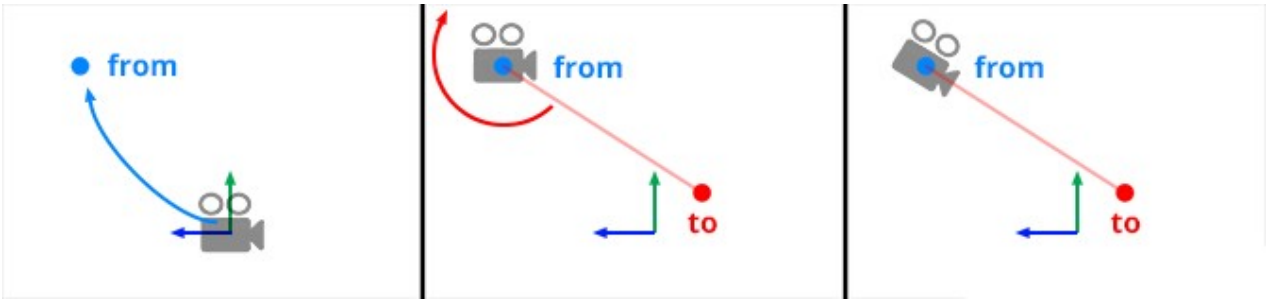


Il·lustració 5.6: Es pot moure la càmera a voluntat per l'espai per poder renderitzar la imatge des de la perspectiva desitjada. La posició final de la càmera es pot representar amb una matriu 4x4. Si sabem O (l'origen de la càmera) i P (la posició del píxel en funció del sistema de coordenades de la càmera), podem aconseguir fàcilment O' i P' multiplicant O i P per la matriu «camera-to-world». Finalment podem calcular la direcció del raig: P' – O'.

5.3.1. El mètode «Look-at»

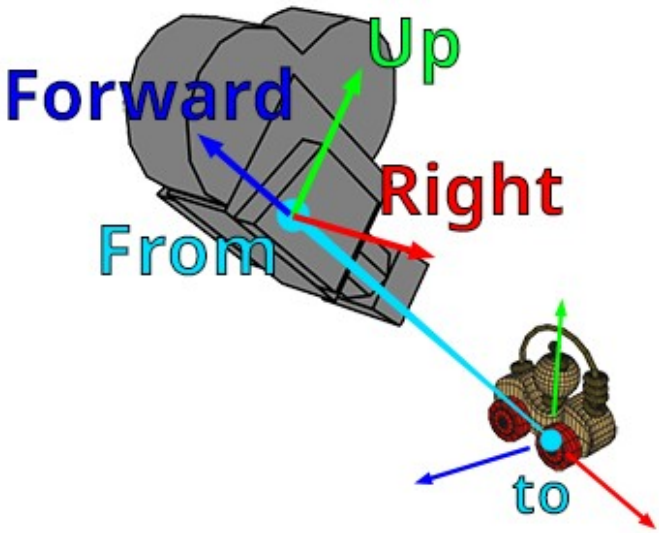
Preparar una matriu per fer totes aquestes transformacions sobre la càmera però, no és una cosa gaire trivial, a no ser que disposem d'algun programa d'animació 3D com per exemple Maya o Blender per configurar la càmera i exportar-ne la matriu de transformació.

Existeix un mètode que és millor a l'hora de crear aquesta matriu «camera-to-world» que crear-la a mà i que no necessita un programa d'animació. No té nom com a tal però sol ser referida com el mètode «Look-at» (o en català «mira cap»). La idea d'aquest mètode és simple. Per canviar la posició i orientació de la càmera només necessitem establir el punt en que es vol que hi hagi l'origen de la càmera, al que ens referirem com a «from» i un altre punt que definirà la posició a la que es vol que miri la càmera, al que ens referirem com a «to». Amb només aquests dos punts ja podem crear la matriu 4x4 que ens servirà per moure de lloc la càmera.



Il·lustració 5.7: El punt «from», on volem que situï la càmera, i el punt «to» al que volem que apunti.

El nom que es posa als eixos d'un sistema de coordenades Cartesià és preferència de cadascú, normalment porten els noms x, y i z, però en aquest apartat els anomenarem «right» (eix x), «up» (eix y) i «forward» (eix z). Ho podem veure visualment en la il·lustració 5.8.



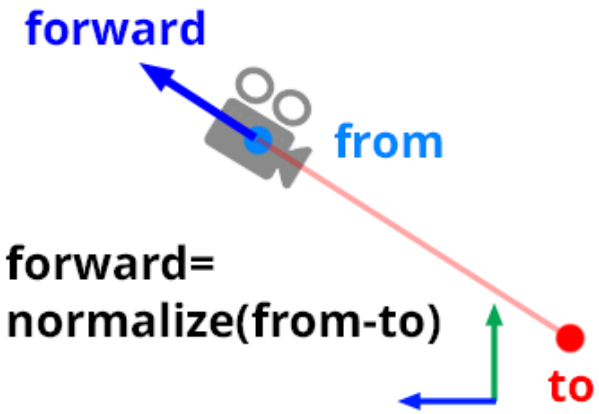
Il·lustració 5.8: El sistema de coordenades d'una càmera.

Aquest mètode és pot dividir en quatre passos:

- **Pas 1: Calcular l'eix «forward».**

  En les il·lustracions 5.8 i 5.9 és fàcil veure que l'eix «forward» està alineat amb la línia que formen els punts «from» i «to». Per calcular aquest eix només normalitzar el vector resultant de $\text{from} - \text{to}$

```
Vec3f forward = normalizeVec3f(subVec3f(from,to));
```



Il·lustració 5.9: Càlcul de l'eix «forward» a partir de la posició de la càmera i del punt al que apunta.

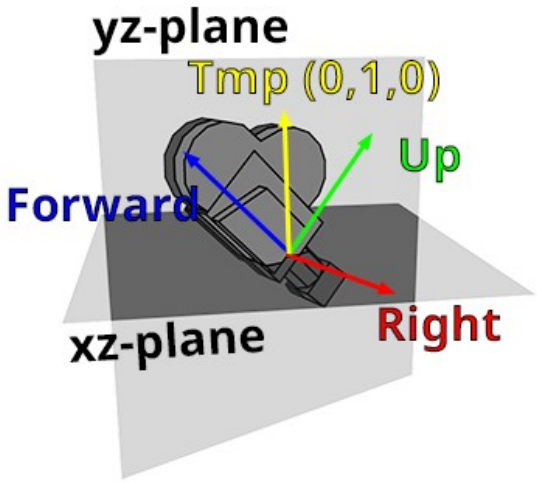
- **Pas 2: Calcular l'eix «right».**

  En el punt 4 (Geometria) hem vist que el producte vectorial entre dos vectors crea un tercer vector que és paral·lel a aquests dos. Es pot fer servir aquesta propietat per crear l'eix «right». La idea és fer servir un *vector arbitrari* per calcular el producte vectorial entre el vector «forward» i aquest vector arbitrari. El resultat és un vector inevitablement perpendicular al vector «forward» que es pot fer servir per construir el sistema de coordenades Cartesià de la càmera com a eix «right».

```
Vec3f right = crossVec3f(randomVec,forward);
```

  Aquest *vector arbitrari* però, no pot ser totalment arbitrari. Es pot veure d'aquesta manera: si el vector «forward» és $(0,0,1)$, aleshores el vector «right» hauria de ser $(1,0,0)$. Això només es pot aconseguir si el vector arbitrari és $(0,1,0)$, de manera que: $(0,1,0)\times(0,0,1)=(1,0,0)$. Mirant a la il·lustració 5.10, es pot veure que sense importar la direcció del vector «forward», el vector perpendicular al pla definit pel vector «forward» i el $(0,1,0)$ sempre és el vector «right» del sistema de coordenades Cartesià de la càmera. Per tant, clarament podem fer servir el vector $(0,1,0)$ com el vector que abans hem anomenat *vector arbitrari*.

```
Vec3f tmp = makeVec3f(0,1,0);
Vec3f right = crossVec3f(normalizeVec3f(tmp),forward);
```



Il·lustració 5.10: El vector (0,1,0) està ubicat al pla definit per vector «forward» i el vector «up». El vector perpendicular a aquest pla és el «right».

- **Pas 3: Calcular l'eix «up».**

  Com que ja tenim els altres dos vectors ortogonals: el «forward» i el «right», només hem de fer el producte vectorial entre aquests dos vectors i aconseguirem l'eix que ens falta («up»). Si els vectors «forward» i «right» ja estan normalitzats, el vector resultant del producte vectorial «up», també estarà normalitzat.

  ```
  Vec3f up = crossVec3f(forward,right);
  ```

  El producte vectorial no és una operació commutativa, així que s'ha d'anar amb compte a amb l'ordre dels vectors a l'hora de fer l'operació.

- **Pas 4: Preparar la matriu 4x4 fent servir els vectors «right», «up» i «forward» i el punt «from».**

  Tot el que queda per fer és construir la matriu «camera-to-world». Per fer-ho, substituirem les files de la matriu per les dades corresponents:

  - Fila 1: canviar els tres primers coeficients de la fila per les coordenades del vector «right».
  - Fila 2: canviar els tres primers coeficients de la fila per les coordenades del vector «up».
  - Fila 3: canviar els tres primers coeficients de la fila per les coordenades del vector «forward».
  - Fila 4: canviar els tres primers coeficients de la fila per les coordenades del punt «from».

  Això és així d'acord amb el contingut del punt 4 (Geometria).

La funció implementada per calcular aquesta matriu «camera-to-world», està definida en el mòdul `matrix`:

```
void makeCamToWorld(Vec3f from, Vec3f to, Matrix44f camToWorld)
```

Pren com a arguments els punts «from» i «to» i una altre variable tipus `Matrix44f` que és on es guardarà la matriu «camera-to-world» ja que la funció no retorna cap valor.

5.3.2. Limitació del mètode «Look-at»

El mètode «Look-at» és simple i generalment funciona bé, de totes maneres, té un punt feble. Quan la càmera està apuntant directament cap amunt o cap avall, l'eix «forward» queda molt a prop del vector arbitrari fet servir per calcular l'eix «right». El cas extrem es dona quan l'eix «forward» i el vector arbitrari són perfectament paral·lels; quan l'eix «forward» és $(0,1,0)$, o $(0,-1,0)$. Desafortunadament, en aquest cas en particular, el producte vectorial falla en crear l'eix «right», i de fet no hi ha cap solució per aquest problema. El que s'hauria de fer és detectar aquest cas i escollir els vectors manualment.

5.4. Funció «render»

El codi font de tot el que hem vist en aquest apartat està en el mòdul `ppm_image`, en la funció `render`. El codi recorre en bucle tots els píxels de la imatge i calcula un raig per cada píxel. Finalment, es fa una crida a la funció `castRay`, que llança el raig cap a l'escena i ens retorna el color corresponent del píxel en qüestió. Mes endavant, en el punt 7, veurem la implementació d'aquesta funció.

```
static void render(Scene *scene, Bitmap *data) {
  Options *options = &scene->options;
  int width = options->width;
  int height = options->height;
  float fov = options->fov;

  float scale = tan((fov/2)*M_PI/180);
  float imageAspectRatio = width / (float)height;
  Vec3f orig = multVecMatrix(VEC3F_ZERO, options->cameraToWorld);
  for(uint32_t j=0; j<height; j++) {
    for(uint32_t i=0; i<width; i++) {
      //raster space to screen space
      float x = (2*(i+0.5)/(float)width-1) * imageAspectRatio * scale;
      float y = (1-2*(j+0.5)/(float)height) * scale;

      Vec3f P = {x,y,-1};
      Ray ray = { .origin = orig,
          .direction = normalizeVec3f(multDirMatrix(P, options->cameraToWorld))
};
      Color color = castRay(&ray, scene, 5);
      draw(data, color, i, j);
    }
  }
}
```

Les opcions de la càmera com les dimensions de la imatge que es vol renderitzar, el camp de visió o la matriu «camera-to-world» es guarden en un `struct` anomenat `Options` definit en el mòdul `scene`.

```
typedef struct {
  int width;
  int height;
  float fov;
  Matrix44f cameraToWorld;
} Options;
```

Aquest `struct` d'opcions forma part de l'escena on s'hi guarden tots els objectes i llums que és del que se'n vol obtenir una renderització.

6. OBJECTES I INTERACCIONS

En aquest apartat es presenta l'abstracció d'objectes com esferes o cilindres i les funcions implementades per tal de comprovar si un determinat raig hi col·lisiona. Una minuciosa abstracció d'aquests objectes i les seves funcions de col·lisió és determinant en un ray tracer per tenir un disseny net.

Hi ha dos parts a tenir en compte per representar un objecte: la seva forma i el seu material. En la forma hi trobem la informació geomètrica de l'objecte; com la seva posició o les seves dimensions. Per altre banda, la part del material ens indica, per exemple, de quin color és l'objecte o quin és el seu índex de reflexió.

La definició d'objecte es troba en el mòdul `surface` i agrupa la forma (informació geomètrica) i el material que defineixen un objecte:

```
typedef struct {
  enum GeometryType type;
  union Geometry geometry;
  Material material;
} Surface;
```

L'struct que defineix un objecte té el nom de *Surface* i està format per tres variables: *type*, *geometry* i *material*. *type* i *geometry* s'encarreguen de descriure'n la forma i *material* s'encarrega de descriure'n el material. Més endavant veurem amb més detall aquests tipus de variables.

Hi ha varies funcions implementades per facilitar la creació d'aquests objectes. Per exemple, per un pla o una esfera:

```
Surface surface_initSphere(Vec3f center, float radius, Material material);
Surface surface_initPlane(Vec3f center, Vec3f normal, Material material);
```

Aquestes funcions retornen un struct del tipus *Surface* i prenen com arguments totes les dades necessàries per representar el respectiu objecte.

6.1. Bases d'una forma

Les formes estan definides en el mòdul `geometry` i cada una (cilindre, esfera, pla...) té un struct amb el seu nom que conté la seva informació geomètrica. Aquest és l'struct base que aporta la informació necessària per identificar la forma de l'objecte i per saber-ne la posició.

A part d'aquest struct, hi ha definits dos altres tipus de dades per facilitar la implementació d'un objecte: un *enum* per identificar de quin tipus d'objecte es tracta i un *union* que agrupa tots els tipus de formes en un de sol. Aquests dos tipus de dades son els que apareixen en la implementació d'objecte (*Surface*) que hem vist el en punt anterior, que es correspondrien amb les variables *type* i *geometry*.

```
enum GeometryType {
  GTSphere,
  GTPlane,
  GTDisk,
  GTCylinder
};

union Geometry {
  Sphere sphere;
  Plane plane;
  Disk disk;
  Cylinder cylinder;
};
```

Un detall important és que s'ha fet servir un *union* per agrupar tots els tipus de formes i no algun altre tipus de dada. El tipus *union* ens permet emmagatzemar diferents tipus de dades en el mateix espai de memòria, de manera que només fa servir l'espai equivalent a un dels seus membres cada vegada que l'utilitzem. En canvi, un *struct* reserva espai per cada un dels seus membres, de manera que cada vegada que volguéssim representar un objecte també estaríem reservant memòria per cada una de la resta de formes implementades.

En aquest cas el tipus *union* tampoc és perfecte però. La memòria que ocupa un *union* es correspon amb la del membre més gran que conté. Per exemple, de les quatre formes que hi ha implementades la que menys ocupa és l'esfera amb 20 bytes i les que més el cilindre i el disc, que ocupen 28 bytes. Encara que representem una esfera (20 bytes), el tipus *union* ocuparà 28 bytes, de manera que estem malgastant 8 bytes. En cas de fer servir un *struct* però, encara seria pitjor, ja que per qualsevol forma que representéssim ocuparíem 100 bytes (el pes de totes les formes representades) i per tant es malgastaria molta més memòria.

Una altre manera d'implementar-ho seria fent us de *pointers*, però la complexitat que implica aquesta implementació no surt a compte en aquest cas.

6.1.1. Interacció raig-forma

Per cada forma implementada també hi ha una funció per comprovar si un raig amb origen i direcció determinades hi impacta. Aquestes funcions es troben definides en el mòdul `rays`.

```
static bool intersectSphere(const Ray *ray, const Sphere *sphere, float *t);
static bool intersectPlane(const Ray *ray, const Plane *plane, float *t);
static bool intersectDisk(const Ray *ray, const Disk *disk, float *t);
static bool intersectCylinder(const Ray *ray,const Cylinder *cylinder,float *t)
```

Aquestes funcions prenen com a paràmetres el raig en qüestió, el *union Geometry* d'on han de treure les dades geomètriques per fer les comprovacions i un apuntador a *float*. En cas que es produeixi un impacte la funció retornarà *true* i guardarà la distància que hi ha entre l'origen del raig i la forma a l'apuntador *float* que se li ha passat com a paràmetre, en cas que no es produeixi cap impacte, la funció retorna *false*. En els següents apartats es veurà amb detall la implementació de cada una d'aquestes funcions.

Aquestes funcions però, estan declares com *static*, de manera que només són visibles dins del mateix mòdul `rays`. Això és així perquè hi ha una altre funció implementada que si que es visible fora del mòdul `rays` que ens facilita la comprovació de si un raig impacta contra un objecte:

```
bool checkIntersection(const Ray *ray, const Surface *surface, float *distance)
{
  switch(surface->type) {
  case GTSphere:
    return intersectSphere(ray, &surface->geometry.sphere, distance);
    break;
  case GTPlane:
    return intersectPlane(ray, &surface->geometry.plane, distance);
    break;
  case GTDisk:
    return intersectDisk(ray, &surface->geometry.disk, distance);
    break;
  case GTCylinder:
    return intersectCylinder(ray, &surface->geometry.cylinder, distance);
  default:
    break;
  }
  return 1;
}
```

La funció *checkIntersection* ens permet comprovar si un raig impacte contra un objecte sense haver-nos de preocupar de quin tipus d'objecte es tracta (esfera, cilindre...). Això, juntament amb la implementació de l'*struct Surface* és de vital importància, ja que separen de la resta del sistema la implementació dels objectes i les seves funcions de col·lisió. D'aquesta manera, per la resta del sistema tots els objectes són tractats per igual: el crees amb la seva funció *init*, l'afegeixes a l'escena i ja pots comprovar amb la funció *checkIntersection* si hi col·lisiona algun raig, sense importar de quin tipus d'objecte es tracti.

6.2. Esferes

Les esferes són un cas especial d'un tipus general de figures anomenades quàdrics, que són figures descrites per polinomis quadràtics en x, y, i z. Els quàdrics són el tipus més simple de figura corbada, per tant, són un bon punt de partida per a la integració d'objectes en un ray tracer. En aquest Ray Tracer hi ha implementats quatre tipus de quàdrics: esferes, cilindres, plans i discs.

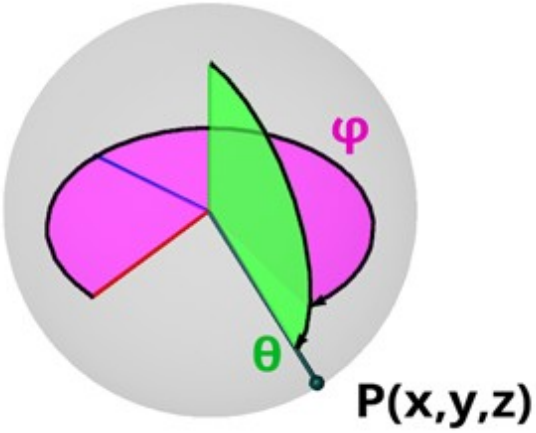
Moltes figures poden ser descrites, generalment de dues maneres: amb la forma implícita o amb la forma paramètrica. Amb la forma implícita, es descriu una superfície 3D com

$$f(x,y,z)=0.$$

El conjunt de tots els punts (x,y,z) que compleixen aquesta condició defineix una figura. Per una esfera amb radi 1 i centrada a l'origen, la seva equació implícita és $x^2+y^2+z^2-1=0$. Només el conjunt de punts situats a una unitat de l'origen satisfà aquesta condició, sent els que estan situats a la superfície de l'esfera.

Moltes figures també es poden descriure amb una forma paramètrica fent servir una funció per passar de punts en 3D a 2D situats a la seva superfície. Per exemple, una esfera amb radi r pot ser descrita com una funció de coordenades 2D d'una esfera (θ, ϕ), on θ pren valors que van de 0 a π i ϕ de 0 a 2π (imatge 6.1).

$$\begin{aligned} x &= r\ \sin\theta\ \cos\phi \\ y &= r\ \sin\theta\ \sin\phi \\ z &= r\ \cos\theta \end{aligned}$$



Il·lustració 6.1: Definició paramètrica d'esfera

L'important aquí es que una esfera pot ser descrita fent servir un conjunt de tres equacions. En aquestes equacions paramètriques, θ i ϕ en són els paràmetres. Els objectes en 3D que poden ser descrits utilitzant aquestes equacions s'anomenen figures paramètriques.

En computació gràfica, aquesta representació és útil, ja que els paràmetres θ i ϕ es poden utilitzar com a coordenades de textura d'un punt en una superfície 3D.

La implementació de l'esfera és la següent:

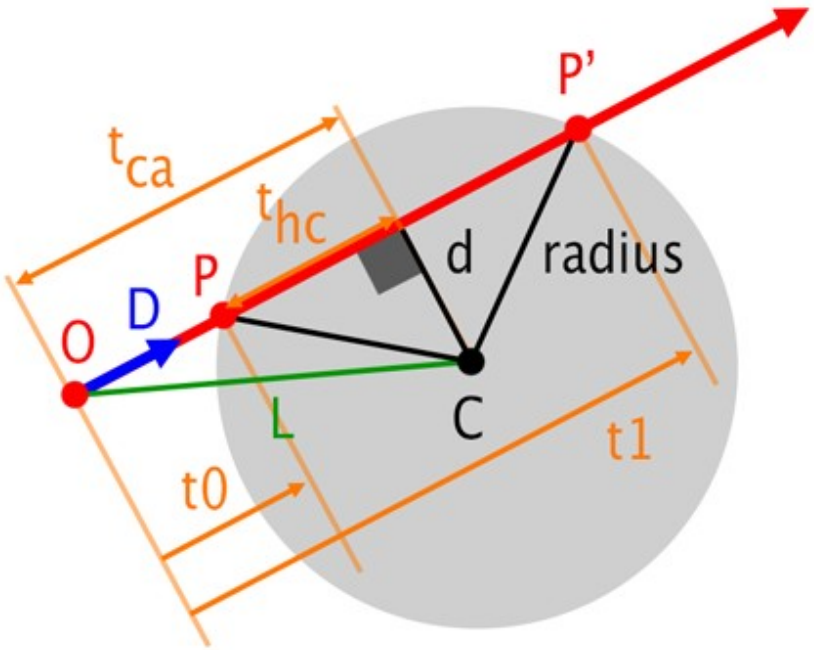
```
typedef struct {
  Vec3f center;
  float radius;
  float radius2;
} Sphere;
```

La variable *center*, que és del tipus *Vec3f* ens indica la posició de l'esfera, i *radius*, del tipus *float*, ens indica el seu radi. Amb aquestes dos variables en tenim suficient per representar una esfera. L'altre variable, *radius2*, és el radi al quadrat, i no és necessària per representar l'esfera. És una optimització per evitar calcular el quadrat del radi repetides vegades i així agilitzar l'execució del codi.

6.2.1. Intersecció raig-esfera

La intersecció d'un raig amb una esfera és probablement la col·lisió raig-figura més fàcil de comprovar. Aquesta és la raó per la qual molts ray tracers mostren imatges d'esferes.

Per comprovar la intersecció d'un raig amb una esfera farem ús d'una solució geomètrica que es basa en matemàtiques simples. Bàsicament geometria, trigonometria i el teorema de Pitàgores. Mirant a la il·lustració 6.2 podem veure que per trobar els punts P i P', que corresponen als punts on el raig col·lisiona amb l'esfera, necessitem trobar el valor de t_0 i de t_1.



Il·lustració 6.2: Un raig creuant-se amb una esfera i varis termes que es faran servir per resoldre la intersecció raig-esfera.

Com ja havíem vist en el punt 4, un raig es pot expressar amb la seva forma paramètrica:

$$r(t) = origin + t * direction \quad 0 \leqslant t \leqslant \infty$$

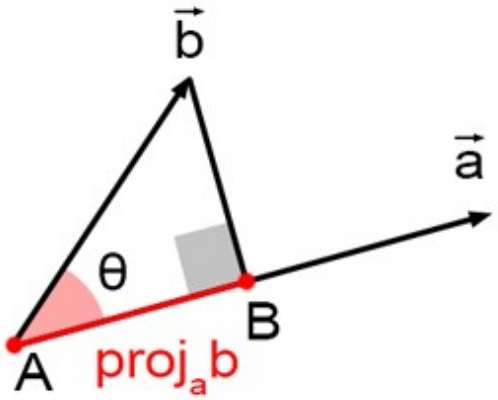
de manera que canviant el valor de t podem aconseguir qualsevol punt pel que passa el raig. Quan t és més gran que 0, el punt està ubicat davant del raig, quan t és igual a 0, el punt coincideix amb l'origen del raig, i quan t és negatiu el punt està darrere l'origen del raig. Mirant a la il·lustració 6.2, podem veure que $t_0 = t_{hc} - t_{ca}$ i $t_1 = t_{hc} + t_{ca}$. Per tant, hem d'aconseguir els valor de t_{hc} i t_{ca} per poder trobar els valors de t_0 i t_1, amb els quals podrem calcular P i P' fent servir l'equació paramètrica del raig:

$$P = O + t_0 D$$
$$P' = O + t_1 D$$

On O representa l'origen i D la direcció.

Podem començar veient que el triangle format pels costats L, t_{ca} i d és un triangle rectangle. L és fàcilment calculable, ja que és el vector que hi ha entre O (l'origen del raig) i C (el centre de l'esfera). Encara no sabem res de t_{ca}, però podem resoldre aquest problema fent ús de la trigonometria.

Sabem L i sabem D, la direcció del raig. També sabem que el producte escalar entre dos vectors $\vec{b}$ i $\vec{a}$ és correspon a la projecció de $\vec{b}$ sobre la línia definida pel vector $\vec{a}$, i que el resultat d'aquesta projecció és la llargada del segment AB com es mostra a la figura XX.



Il·lustració 6.3: $\vec{a} \cdot \vec{b} = |a|\,|b|\,cos(\theta)$.
$|a|\,cos(\theta)$ és la projecció escalar de $\vec{b}$ en $\vec{a}$.

Per tant, com que la direcció del raig és un vector unitari, el producte escalar entre L i D ens dona t_{ca}. Ens hem d'adonar però, que només pot haver-hi una possible intersecció si t_{ca} és positiva, ja que si és negativa vol dir que L i D tenen direccions oposades i per tant si es produís alguna col·lisió seria darrere de l'origen del raig i per tant, no en faríem cas. Ara ja tenim t_{ca} i L.

$$L = C - O$$
$$t_{ca} = L \cdot D$$
$$si\left(t_{ca} < 0\right) \;\; return\, false$$

Hi ha un segon triangle definit per d, t_{hc} i el radi de l'esfera. Ja sabem el radi de l'esfera i per aconseguir el valor de t_{hc} hem de trobar els valors de t_0 i de t_1. Per trobar-los hem de calcular el valor de d. Recordem que d és el costat oposat del triangle rectangle definit per d, t_{ca} i L. per tant el podem trobar amb el teorema de Pitàgores:

$$costat\, oposat^2 + costat\, adjecent^2 = hipotenusa^2$$

Si substituïm amb els valors que nosaltres tenim:

$$d^2 + t_{hc}^2 = radius^2$$
$$d = \sqrt{L^2 - t_{ca}^2} = \sqrt{L \cdot L - t_{ca} \cdot t_{ca}}$$
$$si\left(d < 0\right) \;\; return\, false$$

Si d és mes gran que el radi de l'esfera, el raig falla i no hi ha col·lisió. Finalment, ja tenim tots els termes necessaris per calcular t_{hc}. Podem tornar a fer servir el teorema de Pitàgores:

$$d^2 + t_{hc}^2 = radius^2$$
$$t_{hc} = \sqrt{radius^2 - d^2}$$
$$t_0 = t_{ca} - t_{hc}$$
$$t_1 = t_{ca} + t_{hc}$$

Com hem mencionat abans, podem determinar si un raig falla al col·lisionar amb una esfera si d és més gran que el radi de l'esfera. Per fer aquesta comprovació però, hauríem de calcular l'arrel quadrada de d^2, que té un cost (el processador tarda una estona en fer el càlcul). Per tant, en comptes de calcular si d és més gran que el radi, calculem si d^2 és més gran que $radius^2$. Per això també guardem $radius^2$ en l'struct de l'esfera.

6.3. Plans

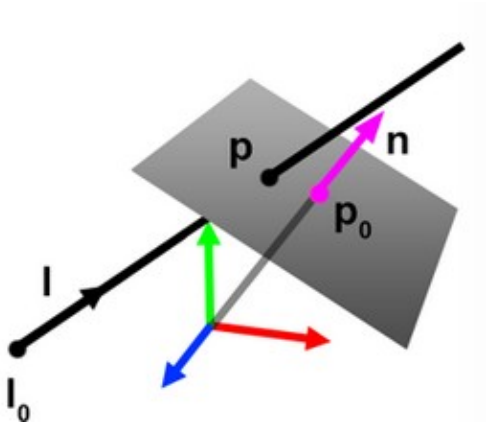
El pla és una figura molt simple. Ve definida per un centre (el centre és relatiu ja que el pla s'esten indefinidament) i un vector normal. Aquest vector determina la orientació del pla.

La definició de pla és la següent:

```
typedef struct {
  Vec3f center;
  Vec3f normal;
} Plane;
```

6.3.1. Intersecció raig-pla

El producte escalar entre dos vectors perpendiculars sempre és igual a 0. Posem p_0 al centre del pla, i n a la seva normal.



Il·lustració 6.4: Intersecció raig-pla.

Es pot generar un vector des de qualsevol punt del pla restant p_0 del punt en qüestió que anomenarem p. El vector calculat està en el pla, de manera que és perpendicular a la seva normal. Per tant:

$$(p - p_0) \cdot n = 0$$

Fent servir la forma paramètrica d'un raig, també es pot dir que:

$$l_0 + l * t = p$$

on l_0 és l'origen del raig i l la seva direcció. Per tant, podem substituir l'equació del raig en l'altre equació:

$$(l_0 + l * t - p_0) \cdot n = 0$$

El que ens interessa d'aquí es aconseguir un valor de t amb el que es podrà calcular un punt d'intersecció fent servir la forma paramètrica d'un raig. Aïllant t s'aconsegueix:

$$t = \frac{(p_0 - l_0) \cdot n}{l \cdot n}$$

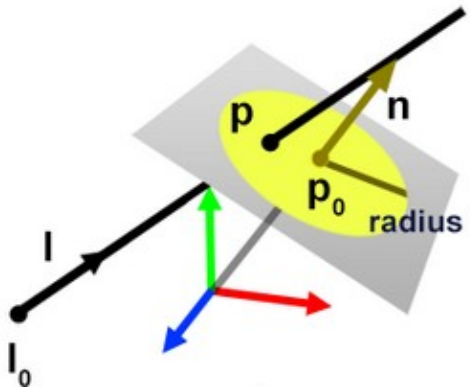
El raig i el pla son paral·lels quan el denominador s'acosta a 0. En aquest cas poden passar dos coses: una es que el pla i el raig s'estiguin tocant, el que provocaria que hi hagués infinites solucions, o que el raig estigui separat del pla, i en aquest cas no es produiria cap intersecció.

6.4. Disc

El disc és casi que el mateix tipus d'objecte que un pla però està acotat per un radi.

La seva implementació és la següent:

```
typedef struct {
  Vec3f center;
  Vec3f normal;
  float radius;
} Disk;
```



Il·lustració 6.5: Intersecció raig-disc

6.4.1. Intersecció raig-disc

La funció d'intersecció d'un disc és molt simple. Podem utilitzar la funció ja creada per comprovar la interacció entre un raig i un pla, i en cas que es detecti una interacció s'ha de calcula la distància que hi ha entre el punt d'intersecció i el centre. Si aquesta distància és igual o menor al radi del disc aleshores hi es produeix una col·lisió, en cas contrari el raig falla.

```
static bool intersectDisk(const Ray *ray, const Disk *disk, float *t) {
  if(!intersectPlane(ray,(Plane*)disk,t))
    return false;

  Vec3f p = addVec3f(ray->origin, multVec3f(ray->direction, *t));
  Vec3f v = subVec3f(p,disk->center);
  float d2 = dotProduct(v, v);
  return sqrtf(d2) <= disk->radius;
}
```

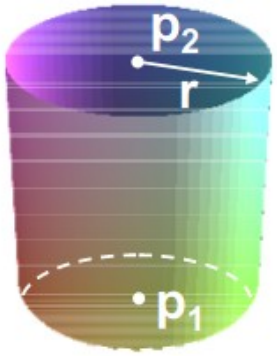
6.5. Cilindres

Els cilindres són una altre de les formes implementada en aquest projecte. Per representar-los, s'han fet servir dos punts, que marquen l'inici i el final del cilindre i un nombre, que ens diu el seu radi.

La seva implementació és la següent:

```
typedef struct {
  Vec3f startPos;
  Vec3f endPos;
  float radius;
} Cylinder;
```

Com es pot veure en la il·lustració 6.4, aquestes variables són suficients per representar-lo en un espai 3D.

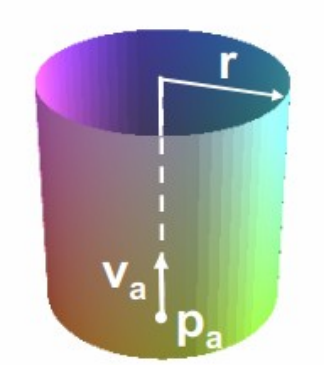


Il·lustració 6.6: Representació d'un cilindre.

El cilindre també es pot veure com la unió de tres figures diferents: dos discs que el limiten i un «tronc» que els uneix.

6.5.1. Intersecció raig-cilindre

Per comprovar si un raig col·lisiona amb el cilindre, primer es farà la comprovació contra un cilindre amb la mateixa orientació però que no estarà acotat entre dos punts. Això és, un cilindre infinit. Fer la comprovació contra aquest cilindre infinit és mes simple. En cas que el raig col·lisioni contra aquest cilindre, s'haurà de comprovar si el punt de col·lisió es troba entre els dos extrems definits.



Il·lustració 6.7: Nomenclatura utilitzada per definir les equacions del cilindre.

L'equació d'un cilindre de radi r orientat sobre una línia $p_a + v_a t$ és:

$$(q - p_a - (v_a, q - p_a) v_a)^2 - r^2 = 0$$

On $q = (x, y, z)$, és un dels punts en el cilindre.

Per trobar els punts d'intersecció amb un raig definit com: $p + vt$, s'ha de fer la substitució $q = p + vt$ i resoldre l'equació:

$$(p + vt - p_a - (v_a, p + vt - p_a) v_a)^2 - r^2 = 0,$$

Això s'acaba reduint a:

$$At^2 + Bt + C = 0$$

Amb:

$$A = (v - (v, v_a) v_a)^2$$
$$B = 2(v - (v, v_a) v_a, \Delta p - (\Delta p, v_a) v_a)$$
$$C = (\Delta p - (\Delta p, v_a) v_a)^2 - r^2$$

On $\Delta p = p - p_a$.

En cas que el raig col·lisioni amb aquest cilindre infinit. S'han de fer més comprovacions per saber si el raig també col·lisiona amb la versió del cilindre acotada.

Primer s'ha de comprovar si el punt de col·lisió es troba entre la mida acotada del cilindre. Després s'ha de mirar si el raig col·lisiona amb algun dels dos discs del cilindre i per últim, s'ha de mirar quin dels punts de col·lisió obtinguts és el que té la distància més petita (t) entre l'origen del raig i el punt de col·lisió.

Per tant, recapitulant, primer s'han de trobar les solucions de t_1 , t_2 per $At^2+Bt+C=0$. Si les solucions existeixen, s'han de marcar com a candidats a intersecció les solucions que no son negatives i per les que $(V_a, q_i - p_1) > 0$ i $(V_a, q_i - p_2) > 0$ on $q_i = p + vt_i$.

També s'ha de calcular els paràmetres t_3 i t_4 pels quals el raig col·lisiona amb els discs superior i inferior del cilindre (el càlcul d'intersecció de raig-disc ja s'ha explicat en un altre punt). Si aquestes interseccions existeixen, marquem com a possibles interseccions les que no son negatives i que compleixen les condicions $(q_3 - p_1)^2 < r^2$ i $(q_4 - p_2)^2 < r^2$ respectivament.

Del conjunt de candidats s'agafa com a bo el que té el valor de t més petit.

La implementació en codi del càlcul d'aquesta intersecció és força llarg i per tant no s'inclou aquí. Es pot trobar en la funció `intersectCylinder` en el mòdul `rays`.

6.6. Materials

El material és el que descriu quin es l'aspecte de l'objecte i quines són les seves propietats físiques. Totes aquestes propietats estan recollides en un struct tipus `Material`. Les definicions de material es poden trobar en el mòdul `material`.

```
typedef struct {
  Color color;
  float reflectivity;
  float indexOfRefraction;
} Material;
```

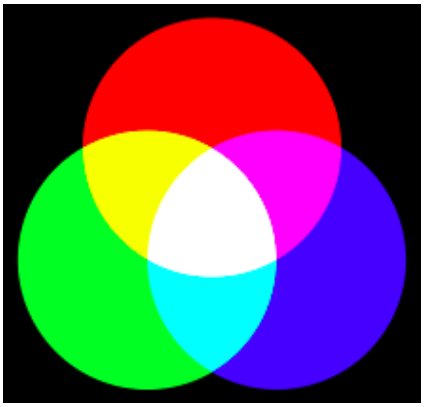
Aquest struct recull el color, l'índex reflectivitat i l'index de refracció de l'objecte.

L'índex de reflectivitat pren valors de 0 a 1, sent 0 gens reflectiu i 1 reflectiu perfecte. L'index de refracció quan és 0 es que es tracta d'un objecte opac i perquè sigui un material transparent, aquest índex ha de prendre un valor més gran que 1.

6.6.1. Color

Una de les propietats d'un objecte és el seu color. El color d'un objecte està establert dins de l'struct `Material`.

La representació del color que es fa en aquest projecte és basa en un sistema RGB.



Il·lustració 6.8: Model additiu de colors vermell, verd i blau.

El model de colors RGB es basa en la representació del color mitjançant una barreja de tres colors de llum primari: vermell, verd i blau.

Color és un struct que guarda tres variables de tipus unsigned char.

```
typedef struct {
  unsigned char r, g, b;
} Color;
```

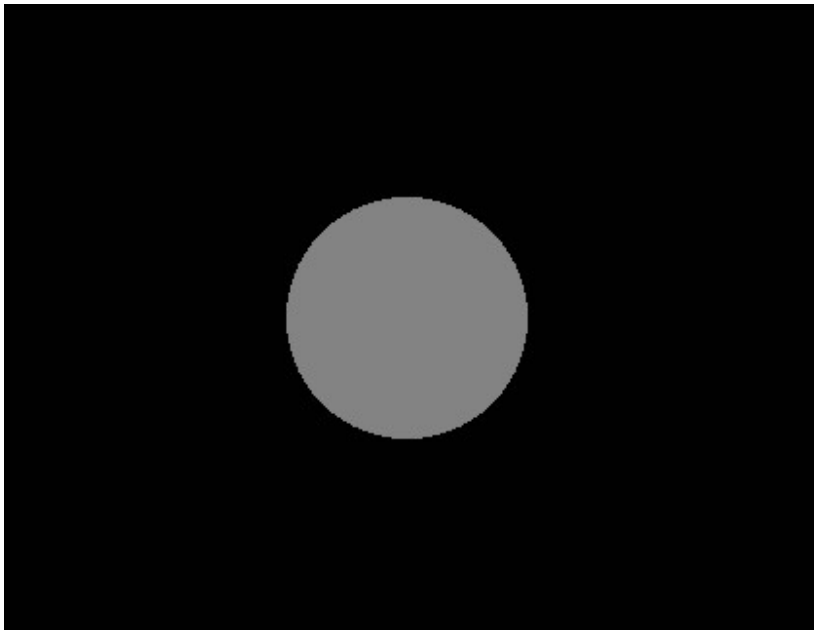
Aquestes variables representen la quantitat de vermell, verd i blau que conté el color que representen. Poden prendre valors que van de 0 a 255, sent (0,0,0) el color negre i (255,255,255) el color blanc (vermell, verd, blau).

Cada pixel de les imatges renderitzades amb aquest ray tracer està format per la combinació de les components d'aquests tres colors.

7. OMBREJAT

Com ja hem dit anteriorment, un procés de renderització pot ser separat en dos passos: visibilitat i ombrejat. La part de visibilitat ja l'hem solucionat en el punt 6, ara veurem l'altre part del procés de renderització: l'ombrejat.

L'ombrejat és el procés que consisteix en calcular els colors dels objectes en una escena en 3D. En un procés de renderització, ens interessa reproduir la forma, la visibilitat i l'aspecte dels objectes vistos des d'un punt de vista determinat. La part de visibilitat s'ocupa de donar-nos la forma de l'objecte i del problema de la visibilitat.



Il·lustració 7.1: Esfera renderitzada només tenint en compte la part de visibilitat.

L'ombrejat s'ocupa de calcular o simular el color dels objectes des d'un punt de vista determinat. El punt de vista té un paper important a l'hora de determinar l'aspecte dels objectes, ja que pot canviar depenent de l'angle de visió. L'aparença d'un objecte també depèn d'altres paràmetres més obvis, com la quantitat de llum que hi ha a l'escena, l'orientació que tenen els objectes en funció de les fonts de llum de l'escena, el color dels objectes... Generalment el perquè les coses es veuen com es veuen té molt a veure en com la llum interactua amb la matèria. Els principis involucrats en aquest procés son coneguts i estan descrits per lleis que generalment són senzilles de simular amb un ordinador. L'ombrejat depèn d'aquestes lleis a l'hora de simular l'aspecte dels objectes d'una manera consistent i realista.

El nivell de realisme del quadre d'un pintor depèn de la seva habilitat per reproduir el color dels objectes vist per l'ull humà en la tela del quadre. Per renderitzar una imatge «foto-realista»,

l'ombrejat ha d'aconseguir un nivell de realisme tal que l'ull humà no pugui distingir si es tracta d'una imatge o del món real.

Il·lustració 7.2: A primera vista aquesta imatge sembla una foto, però en realitat és un quadre. El Simon Hennessey és un pintor que crea quadres que semblen més aviat fotos que pintures.

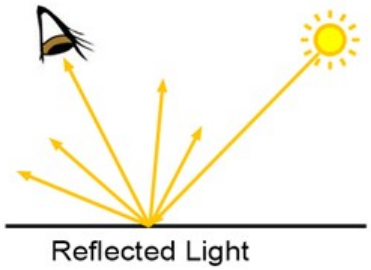
Quan estem generant una imatge amb un ordinador, no podem dependre dels ulls del pintor per decidir de quin color hauria de ser cada píxel de la imatge. Afortunadament, l'aspecte dels objectes, que és el que volem simular amb el renderitzat foto-realista és «només» la combinació de dos coses: la il·luminació i les propietats dels objectes, i aquestes dos coses es poden simular amb un ordinador. No podem veure els objectes si no hi ha llum, així que la llum pren un rol essencial en el procés i quanta més llum hi hagi, més brillants es veuran els objectes. Per altre banda, les propietats dels objectes les podem dividir en dos categories: Les propietats geomètriques de la superfície (com la seva orientació) i les propietats que tenen a veure en com la llum interactua amb un objecte (com per exemple, el seu color).

7.1. Principis físics

En aquest apartat s'explica d'una manera superficial el com i perquè els objectes es veuen de la manera que els veiem: Il·luminació que reben, el color que tenen...

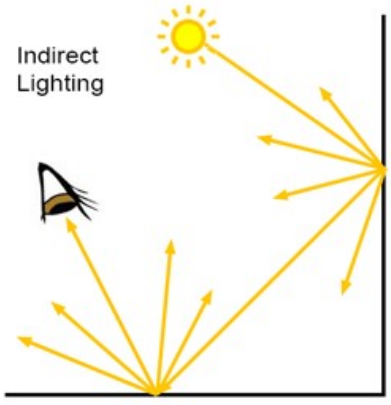
7.1.1. Llum directe i llum indirecte

Quan «veiem» un objecte, el que realment estem veient és la llum reflectida de la seva superfície, ja que la majoria dels objectes no emeten llum. Generalment la llum ve del que en diem fonts de llum; objectes especials que tenen la capacitat d'emetre llum (el sol, les bombetes o la flama d'una espelma en son exemples). Normalment, la llum emesa per una font de llum col·lisiona contra un objecte i part d'aquesta llum és reflectida cap a l'observador (il·lustració 7.3).



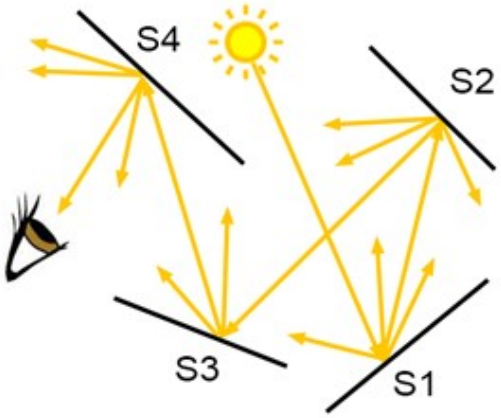
Il·lustració 7.3: La llum es reflectida pels objectes. Part d'aquesta llum ens arriba als ulls i per això veiem els objectes.

En l'ombrejat això és el que anomenem il·luminació directa. Encara que els objectis reflecteixin la llum que els incideix, aquesta no té perquè ser directe, també pot venir indirectament reflectida per la d'algun altre objecte de l'escena. Quan un objecte reflecteix llum cap a un altre objecte, que a continuació és reflectida cap a l'observador (o almenys part de la llum com explicarem a continuació), en diem il·luminació indirecte (il·lustració 7.4).



Il·lustració 7.4: Il·luminació indirecte: la llum reflectida per objectes pot il·luminar altres objectes de l'escena.

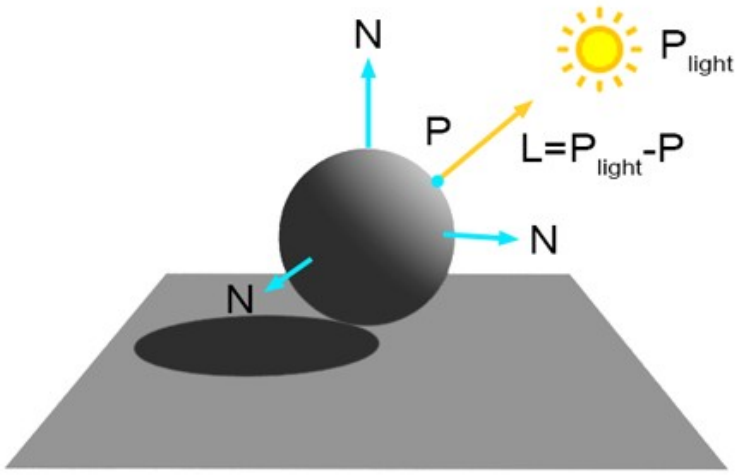
També pot passar que la llum reboti varies vegades abans d'arribar a l'observador (il·lustració 7.5).



Il·lustració 7.5: El nombre d'objectes que poden reflectir la llum abans d'arribar a l'observador en teoria poden ser infinites i/o a la pràctica moltes.

7.1.2. Interacció de llum i matèria

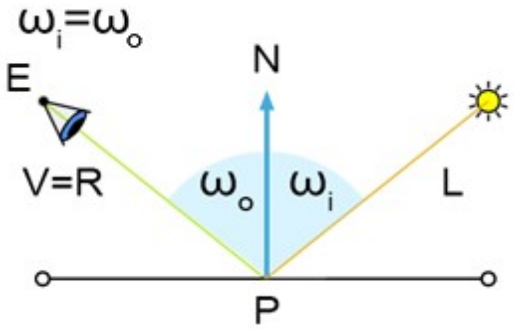
El que hem vist fins ara és que en realitat no veiem els objectes com a tal sinó la llum que reflecteixen. La llum és energia emesa per fonts de llum i reflectida per la superfície dels objectes. Com hem dit anteriorment, l'orientació de les superfícies és un factor que determina la quantitat de llum que reflecteix un objecte; tots sabem que si orientem una superfície cap a una font de llum aquesta es torna més lluminosa. Aquí es on la normal N a la superfície d'un objecte ens serà de més utilitat. La direcció de la llum L és pot definir com la línia (un vector) que va del punt de la superfície P del qual en volem calcular la llum cap a la font de llum (il·lustració 7.6).



Il·lustració 7.6: La lluminositat d'un punt a la superfície d'un objecte depèn de l'orientació de la normal de la superfície en aquest punt en relació amb la direcció del raig de llum.

Sabem que el color en un punt d'una superfície depèn de P, el punt de la superfície del qual en volem saber el color, N, la normal en el punt i L, la direcció de la llum (com de «lluminosa» és una font de llum també és un factor important).

Fins ara hem dit que la llum es reflecteix de la superfície dels objectes, però no hi hem entrat gaire en detall. L'angle que formen el raig de llum incident i la normal de la superfície en el punt d'impacte s'anomena angle d'incidència, denotat amb la lletra grega omega ω. Les superfícies reflectants com un mirall, reflecteixen la llum de la mateixa manera que una pilota de tenis rebota contra la pista. La direcció del raig reflectit també forma un angle amb la normal al punt d'impacte; aquest s'anomena angle de reflexió. L'angle d'incidència i l'angle d'incidència son governats per les lleis de la reflexió, que diuen que quan un raig de llum col·lisiona contra una superfície, l'angle d'incidència és igual a l'angle de reflexió (il·lustració 7.7). Casi tot en el procés d'ombrejat deriva, d'alguna manera, d'aquesta simple llei.



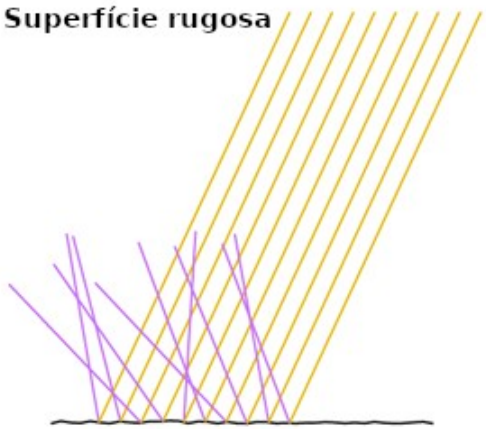
Il·lustració 7.7: L'angle d'incidència i l'angle de reflexió són equivalents si la superfície és un mirall.

Poques superfícies en el món real però, son miralls perfectes. La majoria de superfícies són, de fet, d'alguna manera llustroses, mate o difoses.

Il·lustració 7.8: Tipus de superfícies

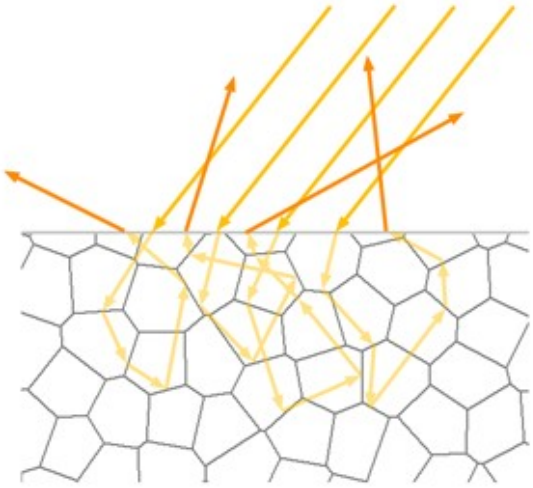
Les superfícies difoses son, per dir-ho d'alguna manera, com miralls trencats. La seva superfície no és perfectament llisa i la podem interpretar com un conjunt de petites cares, totes actuant com petits miralls però orientats en direccions aleatòries (il·lustració 7.9). Com a resultat, la llum, en comptes de reflectir-se com en la superfície d'un mirall, es reflexa en direccions lleugerament

diferents a la ideal que donaria un mirall. Com de lluny s'aparten aquestes direccions de la perfecte d'un mirall depèn de com de diferent és la orientació d'aquestes petites cares en relació a la superfície «perfecte» d'un mirall.



Il·lustració 7.9: Les superfícies difoses són com «miralls trencats». La seva superfície no es perfectament llisa i es pot interpretar com un conjunt de petites cares que actuen com a miralls però que tenen direccions lleugerament diferents.

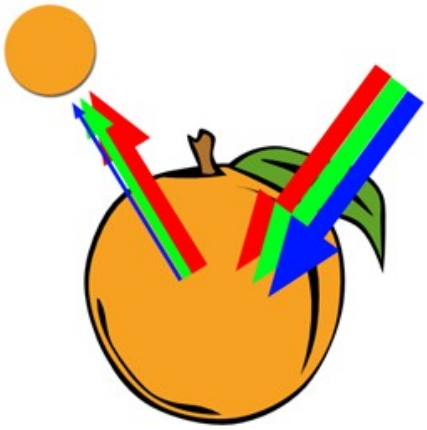
Els materials difosos es poden veure com el contrari a miralls perfectes, i ens pot portar a pensar que són superfícies molt rugoses. Tot i que el perquè és veuen difosos té poc a veure amb la rugositat per si sola. Els materials difosos, són materials que acostumen a tenir una estructura interna complexa. Degut a això, els raigs queden «atrapats» dins l'objecte rebotant varies vegades abans de tornar-ne a sortir, això provoca que la direcció que tenen els raigs al sortir de l'objecte no tingui res a veure amb la direcció amb la que han entrat. A més a més, generalment acaben sortint menys raigs dels que entren, ja que l'objecte en pot absorbir uns quants).



Il·lustració 7.10: Material difós.

Per això veiem que els materials difosos reflecteixen llum homogèniament en totes direccions en un hemisferi centrat en el punt de col·lisió P. Això vol dir que hi ha una diferència fonamental entre les superfícies que són brillants o llustroses i les difoses. Com que els objectes difosos reflecteixen llum equitativament en totes direccions la brillantor amb la que els veiem no canvia en funció del punt de vista. En canvi, amb els objectes brillants no passa el mateix. La imatge que veus reflectida en un mirall va canviant en funció del nostre punt de vista, per això diem que les superfícies brillants són dependents del punt de vista. En canvi, les superfícies difoses són independents del punt de vista. Les pots mirar des de l'angle que vulguis que la brillantor amb que les veiem no canviarà.

L'ultim que ens queda per veure sobre l'ombrejat és perquè els objectes tenen el color que tenen i com el podem simular. La llum «blanca» està formada per tot l'espectre de colors visibles. Quan aquesta llum «blanca» impacta contra un objecte, alguns d'aquests colors són absorbits per l'objecte i els altres són reflectits. L'espectre de colors absorbits per l'objecte és una propietat del material en qüestió. Una taronja, per exemple, absorbeix la major part de colors blaus i reflexa els verds i els vermells, que barrejats formen el taronja (també absorbeix part dels verds i per això és veu taronja i no groc). Així doncs, un material pot absorbir tots o una fracció dels colors. Una taronja absorbeix casi tots els blaus (diguem un 90%), part dels verds (diguem un 40%) i una part molt petita dels vermells (posem un 10%). La llum que no és absorbida és reflectida (10% de blaus, 60% de verds i 90% de vermells).

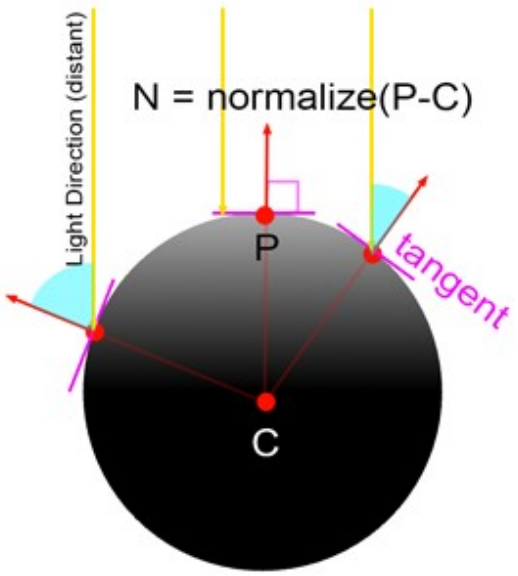


Il·lustració 7.11: La pell d'una taronja absorbeix part dels colors blaus, de manera qe només en reflecteix els verds i els vermells, que barrejats fan el taronja.

Per definir el color d'un objecte primer ens hem d'adonar que el concepte de color només te sentit si parlem d'una superfície que d'alguna manera és difosa. Els miralls no tenen color com a tal. Les taronges en canvi tenen color però també es poden veure una mica brillants. Això és perquè la pell de les taronges, que en gran part és difosa, està coberta per una fina capa d'oli que actua com a mirall. La capa d'oli, que es transparent, reflexa una part de la llum, però també deixa passar una altre part cap a la pell de la taronja que és difosa. El color d'un objecte doncs, es pot definir com la relació de llum reflectida en funció de la llum que rep. En el cas de la taronja, per exemple, aquesta relació seria 0,1 pels blaus, 0,6 pels verds, i 0,9 pels vermells (assumint que és fa servir un sistema RGB per representar els colors.

7.2. Normals

Com ja hem dit en el punt 4.3, el vector normal és molt important en el procés d'ombrejat. Com ja se sap, un objecte es veu més brillant si l'orientem cap a una font de llum, així doncs, l'orientació d'un objecte juga un paper important en la quantitat de llum que reflexa. Aquesta orientació es pot representar en qualsevol punt P de la superfície d'un objecte per la normal N, que es perpendicular a la superfície en el punt P com es veu en la il·lustració 7.12. En aquesta il·lustració també podem veure com la brillantor de l'esfera es va reduint a mesura que l'angle entre el raig de llum i la normal va augmentant.



Il·lustració 7.12: La normal d'un punt en una esfera es pot calcular fàcilment fent servir aquest punt P i el centre de l'esfera.

La complexitat del càlcul de la normal depèn de la forma geomètrica amb la qual estiguem tractant. En aquest projecte s'ha implementat una funció que retorna la normal de l'objecte que li passa com a argument en el punt indicat.

```
void getSurfaceData(const Surface *hitObject, Vec3f Phit, Vec3f *Nhit);
```

Aquesta funció està implementada en el mòdul `surface` i pren com a arguments `hitObject`, que és l'objecte del que en volem aconseguir la normal, `Phit`, que és el punt de col·lisió amb l'objecte i `Nhit`, que és un apuntador a un tipus Vec3f. `Nhit` és on la funció guardarà la normal calculada.

És una funció similar a la funció `checkIntersection` que hem vist en el punt 6.1 en el fet de que serveix per tots els tipus d'objectes. Aquesta funció ja s'encarrega de determinar de quin tipus és l'objecte `hitObject` i crida la funció corresponent pel tipus d'objecte en qüestió.

A continuació veurem la implementació de les funcions particulars per cada tipus d'objecte per aconseguir-ne la normal. Totes aquests funcions estan declarades com a `static` ja que no falta que siguin accessibles fora del mòdul `surface`, ja que com ja hem dit, per aconseguir la normal

d'un objecte en un punt determinat fem una crida a la funció `getSurfaceData` i aquesta ja s'encarrega de fer les gestions necessàries.

7.2.1. Normal d'una esfera

La normal d'una esfera és molt fàcil de calcular. Si sabem el punt de col·lisió i el centre de l'esfera, la normal en el punt es pot calcular restant el centre de l'esfera a la posició del punt de col·lisió (il·lustració 7.12).

```
static void getSurfaceDataSphere(const Surface *hitObject, Vec3f Phit, Vec3f
*Nhit) {
  Vec3f sub = subVec3f(Phit, hitObject->geometry.sphere.center);
  *Nhit = normalizeVec3f(sub);
}
```

Després només hem de normalitzar el vector resultant.

7.2.2. Normal d'un pla i d'un disc

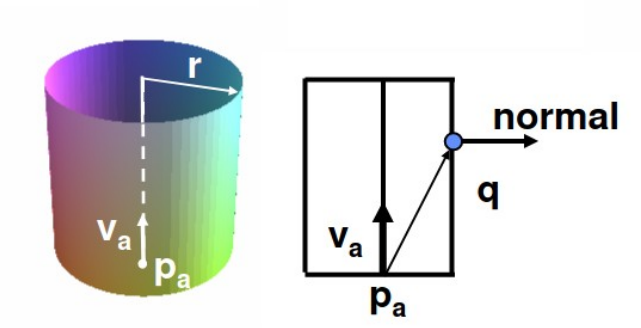
Les normals d'un pla i d'un disc no tenen secret, ja que al ser una superfície plana la normal és la mateixa a tot arreu. L'únic que hem de fer és consultar la normal en les propietats de l'objecte i guardar-lo en la variable `Nhit`.

```
static void getSurfaceDataPlane(const Surface *hitObject, Vec3f Phit, Vec3f
*Nhit) {
  *Nhit = hitObject→geometry.plane.normal;
}

static void getSurfaceDataDisk(const Surface *hitObject, Vec3f Phit, Vec3f
*Nhit) {
  *Nhit = hitObject->geometry.disk.normal;
}
```

7.2.3. Normal d'un cilindre

Com ja hem comentat en el punt 6.5, un cilindre està format per un punt d'inici, un punt de final i un radi. Es podria veure com la unió de tres figures diferents, dos discs i una peça que els uneix.



Il·lustració 7.13: Un cilindre. Pa és el punt d'inici del cilindre, Va es podria veure com la normalització del vector «punt final» - «punt d'inici» i q és el punt de col·lisió del que en volem obtenir la normal.

Per calcular la normal d'un cilindre, primer hem de saber en quina de les tres parts que hem comentat està situat el punt (q) del qual en volem calcular la normal. El punt d'inici i el punt final del disc formen dos discos amb un radi equivalent al del cilindre, per tant, si la distància que hi ha entre el punt de col·lisió q i el centre d'algun d'aquests dos discos és menor que el radi del cilindre voldrà dir que q està situat en un d'aquests dos discs. Per calcular aquesta distància que hi ha entre q i cada centre dels dos discs, hem de fer una resta i calcular-ne el mòdul:

```
float qp1 = modVec3f(subVec3f(Phit,startPos));
float qp2 = modVec3f(subVec3f(Phit,endPos));
```

On `Phit` és el punt q, «`startPos`» es el punt d'inici del cilindre i «`endPos`» el punt final. De manera que `qp1` és la distància que hi ha entre q i el punt d'inici i `qp2`, la distància que hi ha entre q i el punt final.

En cas que algun d'aquest valors sigui menor que el radi, hem de retornar la normal del disc en qüestió, que en aquest cas és el vector Va o $-Va$, que es pot calcular restant el punt d'inici del punt final:

```
Vec3f Va = normalizeVec3f(subVec3f(endPos,startPos));
if(qp1 < radius)
  *Nhit = multVec3f(Va,-1);
else if(qp2 < radius)
  *Nhit = Va;
```

En cas que no estigui situat en cap dels dos disc vol dir que està situat al «tronc» del cilindre, i el càlcul de la normal es pot treure de la següent formula:

$$normal = [(q - pInici) - (Va \cdot (q - pInici))\, Va]_{normalitzat}$$

S'ha de tenir en compte que el punt es refereix a un producte escalar i que s'ha de normalitzar el vector resultant.

```
*Nhit = normalizeVec3f(subVec3f(qp,multVec3f(Va,dotProduct(Va,qp))));
```

Es pot trobar el codi font complet de la funció en el mòdul `surface`.

7.3. Fonts de llum

Una escena està composta per una càmera, llums i objectes. Els llums són una «entitat» especial que ens indica des d'on s'emet la llum que hi ha a l'escena. Com s'ha explicat anteriorment, la raó per la que veiem els objectes és perquè la llum emesa per les fonts de llum rebota de la seva superfície. Si no hi ha llum en una escena, aquesta s'hauria de renderitzar tota de color negre.

Il·lustració 7.14: Dverses fonts de llum

En el món real, qualsevol font de llum té un «cos» físic. Les fonts de llum són objectes que tenen la propietat d'emetre llum, però no deixen de ser objectes comuns que tenen una mida i una forma. Gràcies a això, es poden veure directament amb els ulls, com per exemple les flames d'una foguera, una bombeta o la pantalla d'un ordinador. El problema en el comput gràfic és que representar una font de llum de manera fidel a la realitat amb la seva mida i la seva forma resulta molt costos computacionalment. Per aquesta raó, les fonts de llum en el còmput gràfic s'han estat representant durant molt temps com objectes sense mida ni forma, anomenats «llums delta» (el nom prové de la delta de Dirac o funció d'impuls). Hi ha dos tipus de «llums delta»: les llums distants o les llums esfèriques que veurem en els següents punts.

La implementació de les fonts de llum es troba en el mòdul `lights`, i és molt similar a la manera com s'han implementat els objectes: segueix un sistema d'`structs` que s'acaben agrupant en un de general que representa una llum, sense importar el tipus de llum que sigui. També, igual que els objectes, segueixen una estructura que facilita la implementació de nous tipus de llum al sistema. L'`struct` general que representa un llum és el següent:

```
typedef struct {
  enum LightType type;
  union Lights lights;
} Light;
```

En l'`enum` s'hi troba la nomenclatura que s'ha fet servir per donar nom i poder distingir els diferents tipus de llums:

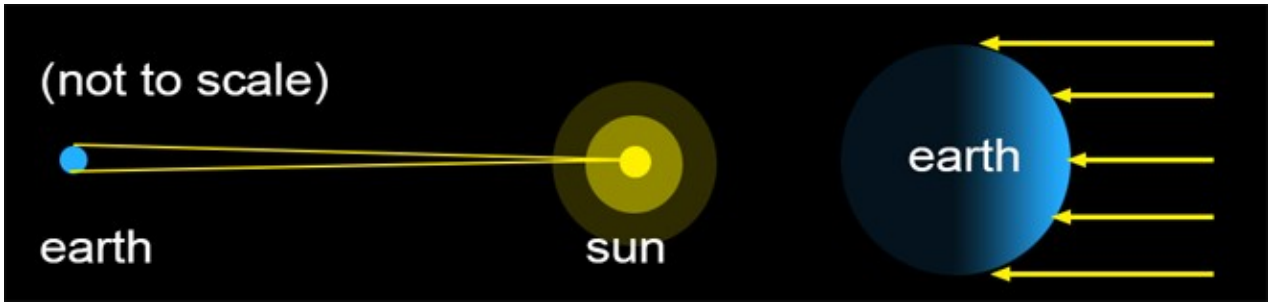
```
typedef struct {
  enum LightType type;
  union Lights lights;
} Light;
```

I el `union` és com el que ja hem vist en la implementació dels objectes. Agrupa els `structs` que defineixen cada tipus de llum.

A continuació veurem els diferents tipus de llum i les seves implementacions.

7.3.1. Llums distants

Les llums distants són les llums considerades tant lluny de nosaltres que la llum que emeten ens arriba en forma de raigs paral·lels entre ells. Amb aquests tipus de fonts de llum l'únic que ens interessa és la direcció d'aquests raigs de llum. Un exemple de llum distant és el sol. El sol és una esfera, que ens porta a pensar que és un tipus de llum esfèrica. A escala del sistema solar o a una escala encara més gran si, es una llum esfèrica, però com es mostra en la il·lustració 7.15, la terra es tant petita comparada al sol i tant llunyana que els raigs de llum que arriben a la superfície de la terra estan continguts en un con super petit de direccions.



Il·lustració 7.15: El sol com a font de llum

Degut a això, es pot considerar que els raigs de llum que arriben a la terra són paral·lels entre ells.

La implementació dels llums distants és simple. Només es necessita una direcció, la intensitat de la llum i el seu color.

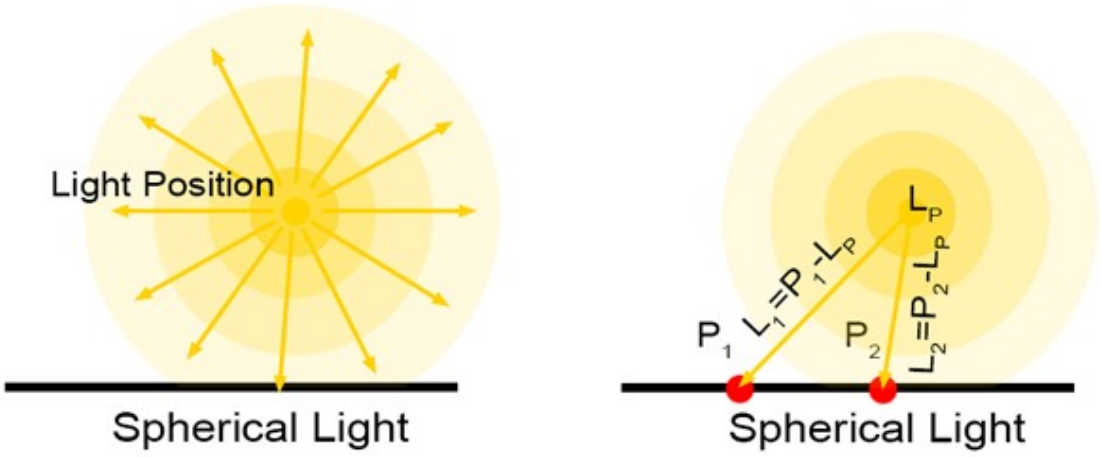
```
typedef struct {
  Vec3f direction;
  float intensity;
  Color color;
} DistantLight;
```

7.3.2. Llums esfèriques

Les llums esfèriques són més complexes que les llums distants. Es poden representar com un punt en l'espai 3D que emet llum. Aquests tipus de llums es poden fer servir per simular la flama d'una espelma, una bombeta, etc. Tot i que es puguin fer servir per fer aquestes simulacions, no són simulacions exactes, ja que la representació que es fa en aquest projecte de llum esfèrica no té volum, no té dimensions. Però de tores maneres, això en aquest projecte no comporta cap inconvenient. Quan una llum esfèrica emet llum equitativament en totes direccions, diem que es isotròpica.

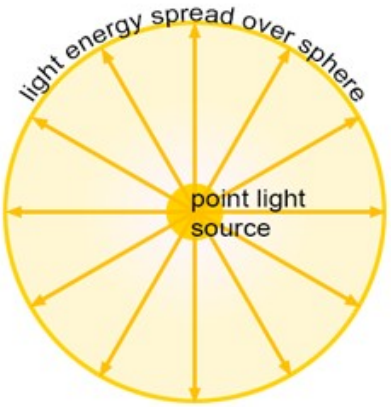
Per simular aquest tipus de llums hem de considerar que la manera que emeten llum és radial. Des del punt de vista de la programació això es pot veure com traçar una línia des del punt que

estem ombrejant P cap a la posició de la llum esfèrica. Aquesta línia ens indica la direcció del raig emès per la font de llum que va cap al punt P (il·lustració 7.16).



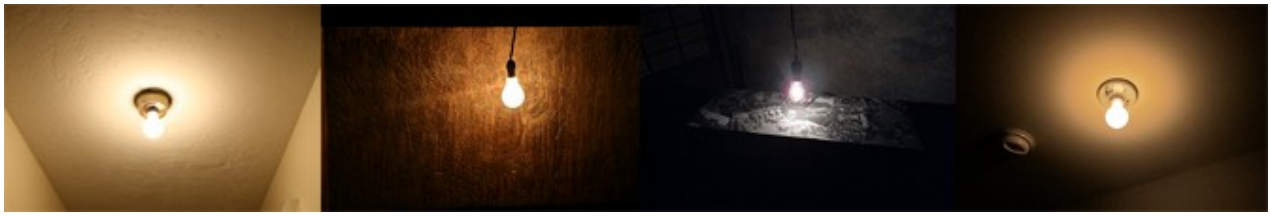
Il·lustració 7.16: Càlcul de la direcció del raig de llum.

Hi ha un altre fenomen a tenir en compte de les fonts de llum esfèriques. Com ja hem mencionat, les llums esfèriques emeten llum en totes direccions des d'un únic punt. L'energia que emeten es distribueix en forma d'esfera com es veu en la il·lustració 7.17. Sense entrar en gaires detalls es considerarà que l'energia del punt lluminós és distribueix en forma d'esfera al voltant d'aquest punt.



Il·lustració 7.17: La llum s'emet des d'un punt i es distribueix en forma d'esfera a voltant del punt.

A mesura que ens anem allunyant del punt però, cada vegada l'energia estarà més dispersada i per tant hi arribarà menys energia, de manera que tot serà més fosc. Aquest fenomen es pot observar amb qualsevol font de llum esfèrica del món real, com per exemple, una bombeta.



Il·lustració 7.18: Exemples de dispersió de llum d'una font de llum esfèrica en el món real.

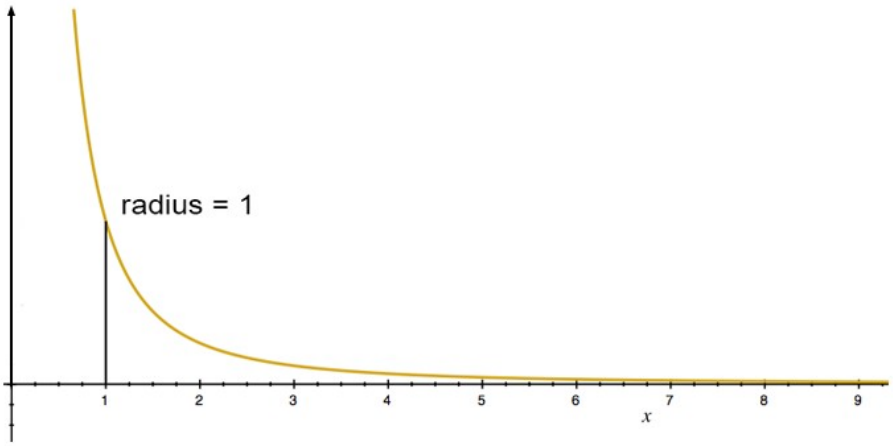
La quantitat de llum que arriba en un punt P de l'escena des d'una font de llum esfèrica depèn de l'àrea d'aquesta «esfera de llum» que es va propagant per l'espai, i aquesta «esfera» depèn del seu radi r :

$$A = 4\pi r^2 .$$

La intensitat de la llum que arriba al punt P és inversament proporcional a l'àrea de l'esfera:

$$L_i = \frac{light\ intensity * light\ \text{color}}{4\pi r^2} .$$

On r és la distància que hi ha entre la posició de la llum i P , i «light intensity» i «light color» són propietats de la font de llum que veurem més endavant. Mirant la formula veiem que r està elevat al quadrat. Això vol dir que quan el radi de l'esfera es fa el doble de gran, l'àrea de l'esfera es multiplica per quatre i per tant la intensitat de la llum és quatre vegades menor. Aquest tipus d'atenuació lluminosa segueix una llei anomenada llei de l'invers del quadrat (il·lustració 7.19).



Il·lustració 7.19: Perfil d'atenuació.

Com es pot veure mirant al gràfic, la intensitat d'un focus de llum s'atenua ràpidament a mesura que incrementem la distància.

La implementació dels llums esfèrics conté els mateixos tipus de variables, però en comptes d'indicar una direcció s'indica la posició del punt.

```
typedef struct {
  Vec3f position;
  float intensity;
  Color color;
} SphericalLight;
```

7.4. Projecció de les ombres

La projecció de les ombres és un problema bastant directe de solucionar, ja que es fa servir el mateix mètode que es fa servir per resoldre el problema de la visibilitat. Ja hem vist que per comprovar els objectes que es veuen des de la càmera, hem de llançar un raig de la càmera cap a l'escena i comprovar si col·lisiona amb algun objecte. Per mirar si en un punt de l'escena hi ha una ombra, el que hem de fer és llançar un raig des d'aquest punt cap a la font de llum. Si aquest raig, anomenat raig d'ombra («shadow ray») col·lisiona amb algun objecte en el seu trajecte cap a la llum, aleshores aquest punt que estàvem mirant està en una ombra.

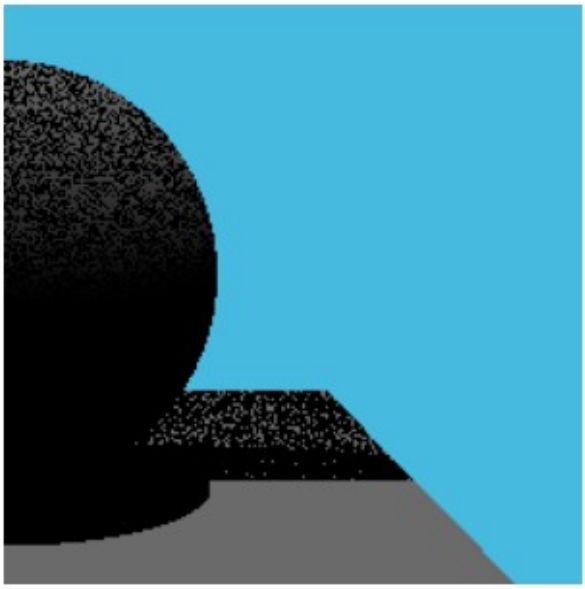
Il·lustració 7.20: Projecció de l'ombra d'una esfera

La funció encarregada de resoldre aquest problema de visibilitat és la funció `trace`, que donat un raig determinat, itera sobre tots els objectes de l'escena i comprova si es produeix una col·lisió amb algun d'ells.

Així doncs, per cada punt de col·lisió derivat dels raigs primaris llençats cap a l'escena, s'ha de crear un nou raig amb origen el punt de col·lisió i direcció a la font de llum. Després s'ha de comprovar si aquest raig col·lisiona amb algun objecte abans d'arribar a la font de llum amb la funció `trace` i retornar el color del pixel que correspongui.

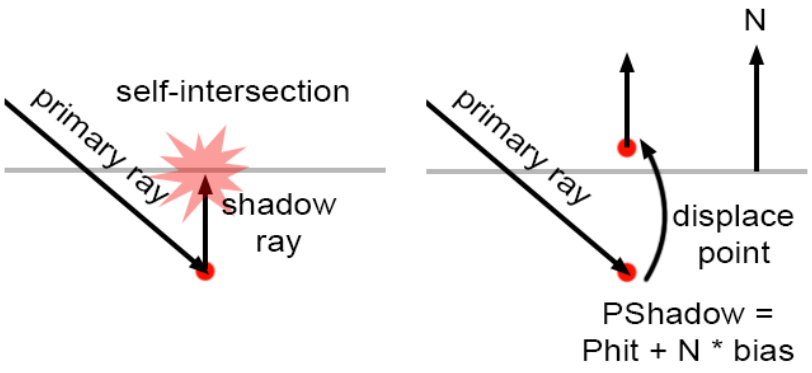
7.4.1. Problema d'auto interacció

El problema d'auto interacció és un problema comú en els ray tracers (il·lustració 7.21). És un efecte visual que apareix en forma de petits punts negres a la superfície dels objectes.



Il·lustració 7.21: Problema d'auto interacció

Això és degut a petits errors introduïts degut a que un ordinador només pot representar els nombres amb un cert nivell de precisió. Això provoca que a vegades el punt d'intersecció s'ubiqui una mica per sota de la superfície. Quan això passa i llancem un «shadow ray» en direcció a la llum, provoca que aquest raig col·lisioni amb la superfície des de la que s'ha llançat (il·lustració 7.22).

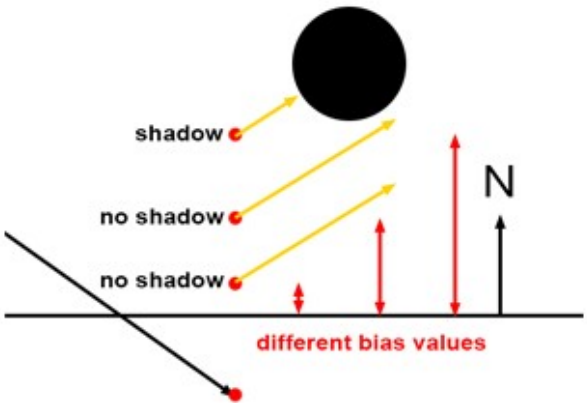


Il·lustració 7.22: El problema de l'auto interacció es degut a un error de precisió numèrica. Degut a aquest error, l'origen del «shadow ray» està sota la superfície, i quan es llança el raig es provoca una auto interacció.

Una solució per aquest problema consisteix en, sistemàticament, desplaçar l'origen dels «shadow rays» en la direcció de la normal de la superfície, de manera que el punt acaba situat damunt de la

superfície. La quantitat de desplaçament que s’aplica ve marcat per un valor anomenat «shadow bias».

El valor que es dona a «shadow bias» sol ser un valor petit, però es pot modificar en funció de l’escena a renderitzar. Un valor correcte, per exemple podria ser 0.0001. Com més gran és el valor de «bias», més es desplaça l’ombra del lloc en el que hauria d’estar.



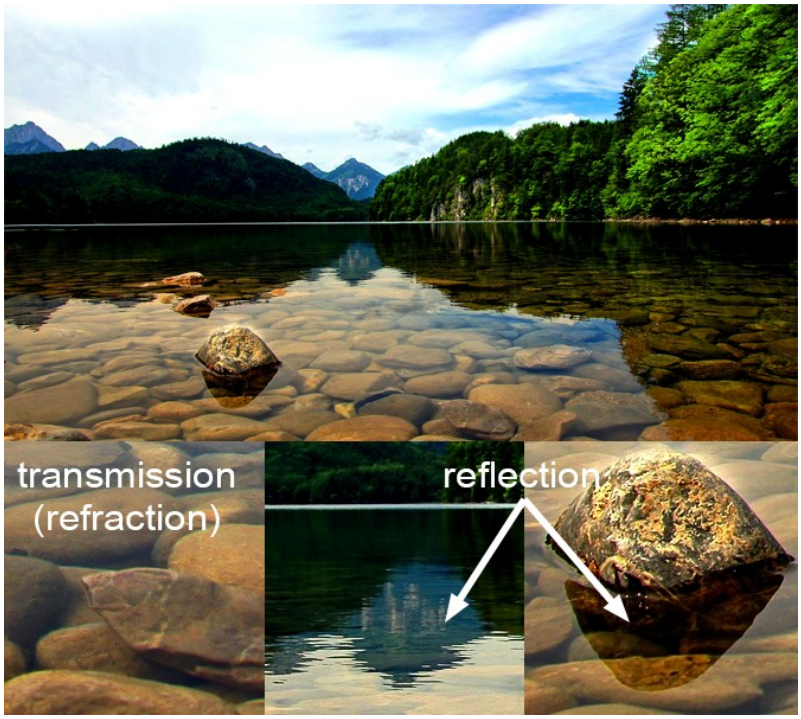
Il·lustració 7.23: Un valor de «bias» massa gran crea una ombra on no n’hi hauria d’haver.

Per aconseguir el valor correcte de «bias», l’hem d’anar canviant i provant. S’ha d’agafar un valor prou gran perquè no es produeixi auto interacció però també ha de ser el més petit possible perquè no canviï de lloc les ombres.

7.5. Reflexió, refracció i Fresnel

La reflexió i refracció són fenòmens que en el món real es poden observar cada dia. L'aigua o el vidre són dos materials que mostren les dos propietats. La llum els pot travessar, un fenomen que en diem transmissió, però també poden reflectir llum al mateix temps. La quantitat de llum reflectida i i transmesa ve determinada per l'efecte Fresel. Hi ha altres materials que són opacs i no poden transmetre llum, però per altre banda, la poden reflectir. Aquest és el cas, per exemple, dels metalls.

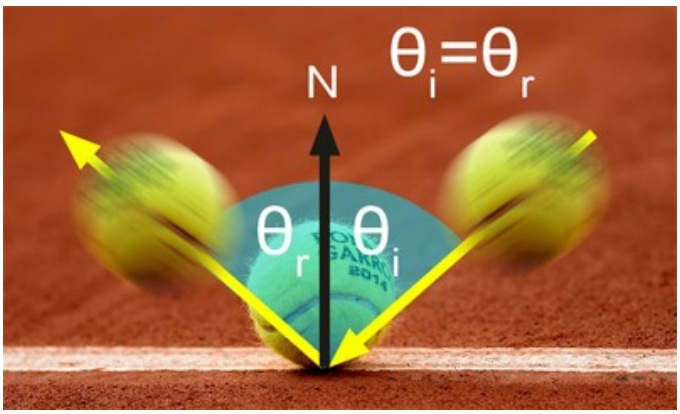
En aquest apartat veurem com simular els efectes de reflexió i refracció i l'efecte Fresnel.



Il·lustració 7.24: Exemple d'efecte de reflexió i refracció.

7.5.1. Reflexió

La reflexió és una des les interaccions de llum amb matèria més simples que existeixen. La reflexió és el que passa quan un fotó (o raig de llum) impacta contra una superfície reflectant, com per exemple vidre, aigua o un tros de paper d'alumini. És molt similar al que li passa a una pilota de tenis quan rebota amb la superfície del terra: rebota en una direcció simètrica a la direcció incident respecte la normal com es mostra en la il·lustració 7.25.



Il·lustració 7.25: L'angle d'incidència i l'angle de reflexió són equivalents.

Calcular la direcció del raig reflectit quan se sap la direcció del vector incident i la normal és molt simple. Com es pot veure en la il·lustració 7.26, els vectors I i R es poden expressar en relació als vectors A i B:

$$I = A + B$$
$$R = A - B$$

El vector B es la projecció del vector I o R en el vector N, que com hem vist en l'apartat 4 (Geometria) es tracta del producte escalar. El vector B doncs, es pot calcular com:

$$B = N \cdot I = \cos(\theta) * N$$

Aleshores es pot substituir B en les dos equacions:

$$I = A + \cos(\theta) * N$$
$$R = A - B\cos(\theta) * N$$

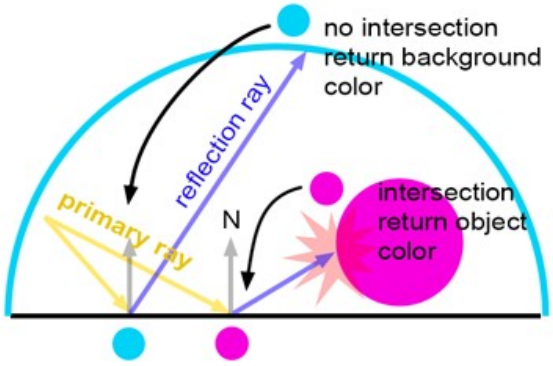
I es pot reescriure la primera equació com:

$$A = I - \cos(\theta) * N$$

I es pot escriure la segona equació fent servir aquest resultat de la següent manera:

$$R = I - \cos(\theta) * N - \cos(\theta) * N$$
$$R = I - 2\cos(\theta) * N$$
$$R = I - 2(N \cdot I) * N$$

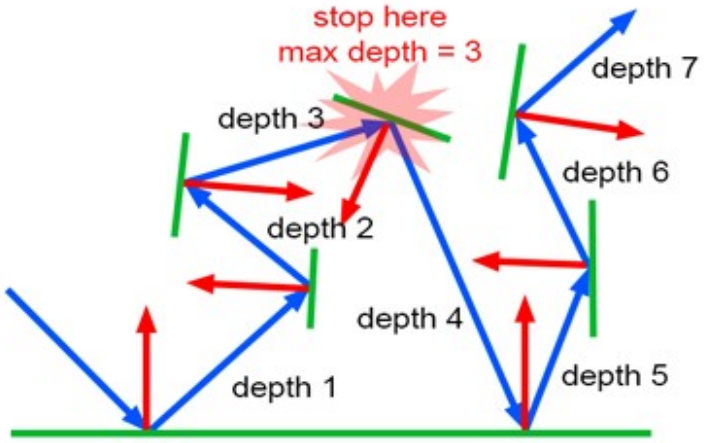
Simular l'efecte de reflexió en un algoritme de ray tracing és simple: si l'objecte amb el que col·lisiona un raig primari és reflectant, aleshores s'ha de calcular la direcció del raig reflectit i tornar a cridar la funció `castRay` recursivament.



Il·lustració 7.26: Si el raig primari impacta contra una superfície que és un mirall, llencem un raig de reflexió. El primer punt de col·lisió prendrà el valor que el raig de reflexió detecti.

Veiem que aquest procés és recursiu, doncs es pot donar el cas en que el raig impacti contra una superfície reflectant, el nou raig que es traça torni a impactar contra una altre superfície reflectant... Per tant, el procés seguira traçant nous raigs a excepció de que un raig col·lisioni amb una superfície que no es reflectant o que no col·lisioni amb res, que en aquest cas retornarà el color de fons.

Si el raig es quedés rebotant en superfícies reflectants sense interaccionar amb algun altre tipus d'objecte, s'entraria en un bucle recursiu infinit. Per evitar que es produeixi aquest cas, s'afegeix un paràmetre anomenat profunditat del raig. Aquest paràmetre posa un nombre de rebots màxims que pot assolir un raig, i el seu valor es va reduint cada vegada que es produeix una reflexió.

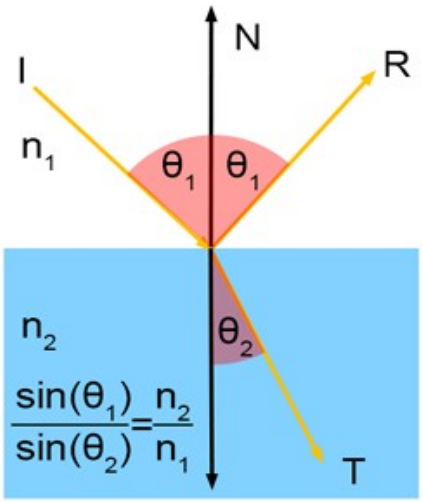


Il·lustració 7.27: La reflexió és un procés recursiu.

Quan es crida la funció `castRay` doncs, primer es comprova el valor d'aquest paràmetre. En cas que el seu valor sigui 0, la funció `castRay` retorna el valor del color de fons sense fer res més, impedint d'aquesta manera entrar en un bucle infinit.

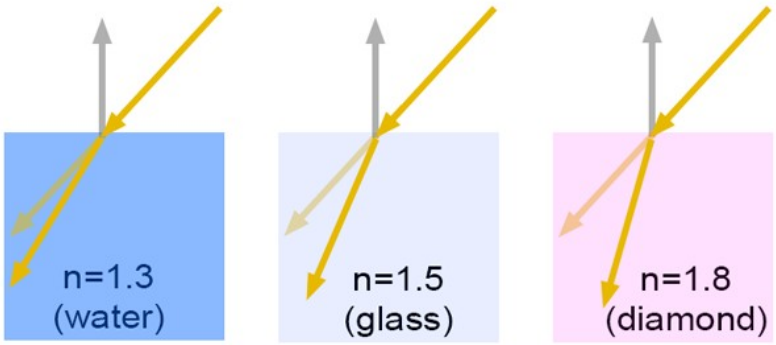
7.5.2. Refracció

Quan els raigs passen d'un medi transparent a un altre, la seva direcció canvia. Aquest fenomen es pot veure en la il·lustració 7.28.



Il·lustració 7.28: Quan un raig passa d'un medi transparent a un altre, la seva direcció canvia.

El canvi de direcció del raig depèn de dos factors: de l'angle d'incidència del raig θ_1 i de l'index de refracció del nou medi (l'index de refracció també rep el nom de ior). L'index de refracció del vidre és, aproximadament 1.5, i el de l'aigua 1.3, així que donat el mateix raig incident pels dos materials, la direcció del raig refractat canvia.



Il·lustració 7.29: Donat el mateix raig incident el raig refractat canvia depenent de l'index de refracció del nou material.

La velocitat de la llum en el buit (c), disminueix si està viatjant a través d'un altre medi. L'index de refracció dels materials ve determinat per la següent equació:

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

On v és la velocitat de la llum en el medi en qüestió, i η el seu índex de refracció. L'index de refracció de l'aire és molt propera a 1, i en aquest projecte no el diferenciarem del buit. La refracció d'un raig de llum explica perquè els objectes es veuen transformats a través d'un medi transparent com l'aigua o el vidre, cosa que pot crear efectes curiosos com la il·lusió d'un llapis trencat dins un got d'aigua.

Il·lustració 7.30: Efecte de la refracció.

La refracció es descriu amb la llei de Snell, que donats dos medis diu el següent:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{\eta_2}{\eta_1}.$$

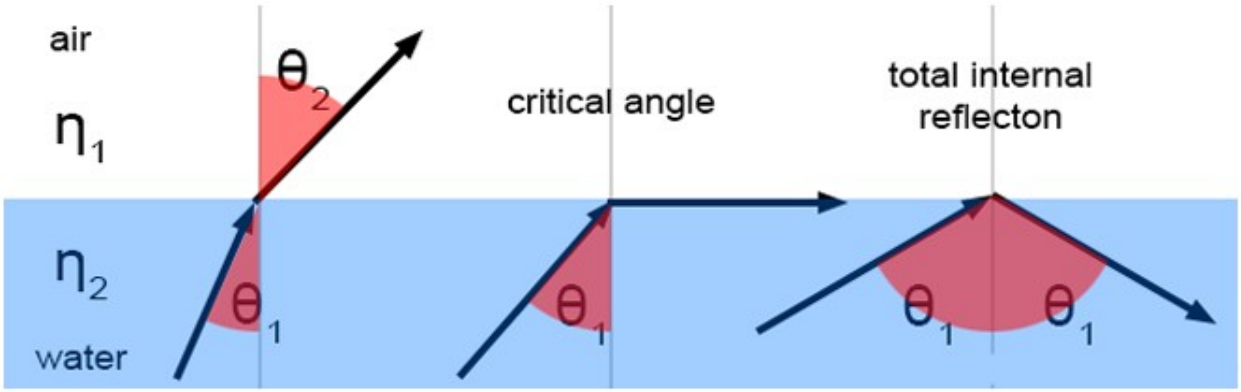
Desenvolupant aquesta equació i el que ja s'havia vist ara per calcular la direcció d'un raig reflectit, obtenim que:

$$T = \eta I + N(\eta c_1 - c_2)$$

On T és el vector refractat, i:

$$\eta = \frac{\eta_1}{\eta_2},$$
$$c_1 = \cos(\theta_1) = N \cdot I,$$
$$c_2 = \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 \sin^2(\theta_1)}$$

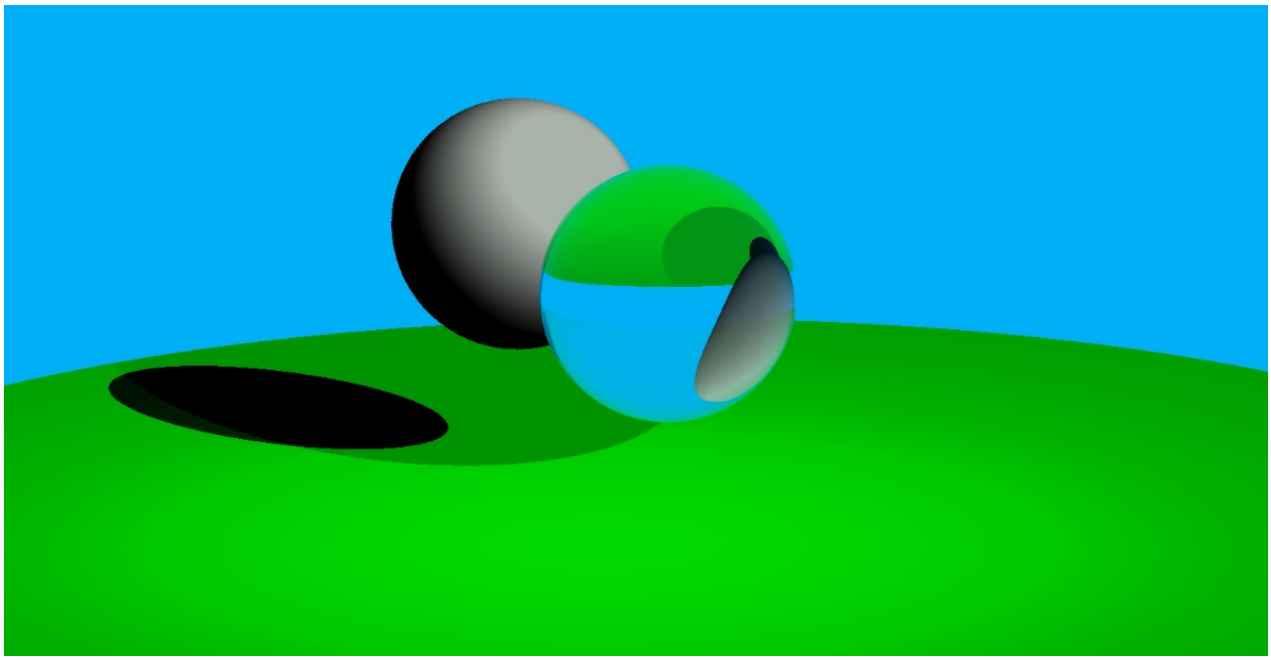
Hi ha un altre aspecte que s'ha de tenir en compte però. Quan l'angle incident és més gran que un valor anomenat angle crític, aleshores el 100% del raig incident és reflectit. Aquest fenomen només es dona quan la llum passa d'un medi a un altre que té un índex de reflexió menor. Com per exemple en el cas que un raig passi de l'aigua a l'aire.



Il·lustració 7.31: Fenomen de reflexió interna total.

Aquest angle és pot calcular, però hi ha una manera més fàcil de saber si es dona el cas de reflexió interna total. Si el terme que hi ha dins de l'arrel quadrada de c_2 és negatiu és que es produeix aquest efecte.

En la il·lustració 7.32, es pot veure una imatge renderitzada amb el ray tracer d'aquest projecte que mostra un exemple del fenomen de refracció.



Il·lustració 7.32: El pla i l'esfera grisa es veuen refractades en l'esfera de vidre.

7.5.3. Fresnel

Com ja s'ha comentat, els objectes produeixen tant reflexió com refracció. La quantitat de llum que en surt reflectida en comparació amb la quantitat de llum que es transmet depèn de l'angle d'incidència. A mesura que l'angle d'incidència és fa més petit, la quantitat de llum transmesa augmenta i viceversa. Seguint el principi de conservació de l'energia, la quantitat de llum reflectida més la quantitat de llum transmesa ha de ser equivalent a la quantitat de llum incident.

La quantitat de llum transmesa i reflectida es pot calcular mitjançant les equacions de Fresnel:

$$F_{R\parallel} = \left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2} \right)^2,$$
$$F_{R\perp} = \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1} \right)^2.$$

(Els termes usats són els que ja s'han comentat a l'apartat anterior.)

Amb la mitjana de les dos equacions aconseguim la proporció de llum reflectida.

$$F_R = \frac{1}{2}(F_{R\parallel} + F_{R\perp}).$$

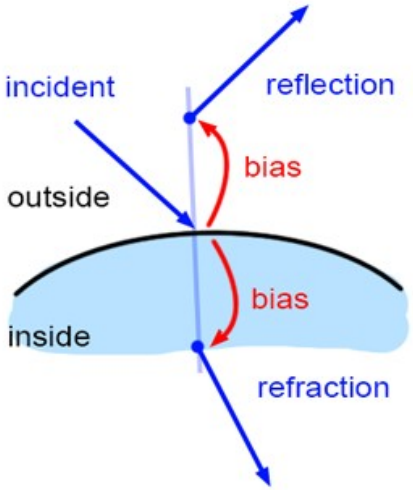
Per altre banda, degut a la llei de la conservació de l'energia, la proporció llum refractada es pot calcular de la següent manera:

$$F_T = 1 - F_R.$$

7.6. Implementació de la funció «castRay»

La funció `casRay` és on hi ha la implementació de tots els conceptes explicats en aquest apartat d'ombreig. Aquesta funció és l'encarregada de determinar el color que retorna un raig determinat.

Hi ha una altre cosa a tenir en compte a l'hora d'implementar la funció. Com abans s'ha comentat, hi ha un fenomen anomenat auto interacció. Aquest fenomen també es pot produir quan es generen els raigs de refracció, per tant, també se'ls hi ha d'aplicar un petit desplaçament.



Il·lustració 7.33: Efecte de l'auto interacció.

Com ja s'ha comentat, en cas que es produeixi una reflexió, l'origen del raig s'ha de desplaçar cap a fora de la superfície, però si el que es produeix és una refracció, l'origen del raig s'ha de moure cap a dintre de la superfície.

A continuació es mostra la implementació de la funció castRay.

```
Color castRay(const Ray *ray, const Scene *scene, size_t depth) {
  if (depth == 0)
    return scene->backgroundColor;
  /* Comprovem si el raig colisiona amb algun objecte */
  TracingResult closestHit = trace(ray,scene, kPrimaryRay);
  if (closestHit.surface == NULL)
    return scene->backgroundColor;

  Material material = closestHit.surface->material;
  Color hitColor = material.color;
  Vec3f Phit = addVec3f(ray->origin, multVec3f(ray-
>direction,closestHit.distance));


  if(material.reflectivity > 0.0 && material.indexOfRefraction == 0.0 && depth
> 0) {
    /* Only reflection */
    Ray reflectedRay = rays_reflect(ray, closestHit.surface, Phit);
    Color reflectionColor = castRay(&reflectedRay, scene, depth-1);
```

```
    hitColor = colorBlend(reflectionColor, material.reflectivity, hitColor);
  }
  else if(material.indexOfRefraction > 0.0 && depth > 0) {
    /* Reflection and refraction */
    Color refractionColor = COLOR_BLACK;
    float kr = fresnel(ray, closestHit.surface, Phit);
    if(kr < 1) {
      Ray refractedRay = refractRay(ray, closestHit.surface, Phit);
      refractionColor = castRay(&refractedRay, scene, depth-1);
    }

    Ray reflectedRay = rays_reflect(ray, closestHit.surface, Phit);
    Color reflectionColor = castRay(&reflectedRay, scene, depth-1);

    hitColor = colorBlend(reflectionColor, kr, refractionColor);
    return hitColor;
  }

  Color lightIntensity = ray_shadeAtPoint(ray, scene, closestHit.surface,
Phit);
  Color highLightedColor = getHighlightedColor(hitColor,lightIntensity);
  if(material.reflectivity > 0.0)   /* Mirrors should only reflect some % of
the shadow */
    return colorBlend(hitColor,material.reflectivity,highLightedColor);
  return highLightedColor;
}
```

8. CONCLUSIONS

La realització d'aquest treball ha estat un procés d'aprenentatge constant. Abans de la realització del projecte ja es es tenien coneixements del llenguatge de programació utilitzat i es tenien certes bases matemàtiques per la seva realització, però tots els conceptes relacionats amb la computació gràfica i el ray tracing eren llunyans i confosos. Conceptes que s'han anat treballant i consolidant durant la realització del projecte.

Els objectius del projecte s'han assolit correctament, ja que s'ha desenvolupat correctament un algoritme de ray tracing i se n'han treballat els conceptes relacionats.

L'evolució del projecte ha estat l'adequada. Es va començar per unes bases ben establertes sobre les que s'ha anat construint i evolucionant per realitzar coses més complicades i complexes. Aquesta metodologia es podria seguir aplicant per anar millorant i ampliant el ray tracer desenvolupat en aquest projecte amb noves funcions i implementacions que li permetessin renderitzar escenes més complexes amb més tipologies d'objectes.

S'ha acabat el projecte amb un renderitzador que és capaç de renderitzar quatre tipus diferents de figures, admet dos tipus diferents de fonts de llum, es poden posar múltiples llums en una escena i també s'han implementat tècniques de renderització que permeten aplicar efectes de reflexió i refracció sobre els objectes.

Es pot concloure doncs, que s'ha aconseguit implementar un algoritme de ray tracing funcional.

Aquest projecte es pot ampliar amb l'addició de funcionalitats i efectes més avançats i complexos, com per exemple un efecte d'oclusió ambiental, l'afegit de textures als objectes, etc. Efectes que aporten més realisme a les renderitzacions.

Actualment encara estan sortint noves tècniques i millores pels algoritmes de ray tracing. Un bon exemple el dona l'empresa Pixer Animation Studios, que cada pel·lícula que treu es veu millor que l'anterior. Per tant, les millores que es poden fer a nivell de realisme i fidelitat de les renderitzacions amb la realitat és molt gran.

Un altre aspecte en el que es podria millorar és en l'eficiència del programa. El programa desenvolupat en aquest projecte, dissenyat amb fins il·lustratius, no està optimitzat per ser eficient. Es podrien implementar noves tècniques o modificar les existents per fer-les més eficients.

9. BIBLIOGRAFIA

[Bra88] Brian W.Kernighan, Dennis M.Ritchie. *The C programming language*. Segona edició. New Jersey: AT&T Bell Laboratories, 1988. ISBN 0-13-110370-9.

[Mat18] Matt Pharr, Wenzel Jakob, Greg Humpherys. *Physically Based Rendering: From Theory To Implementation* [en línia]. 3ª ed. 2018. [Consulta: 7/10/2020]. Disponible a: http://www.pbr-book.org/3ed-2018/contents.html.

[Geo16] Scratchapixel. *Geometry* [en línia]. 2016. [Consulta: 13/08/2020]. Disponible a: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry.

[Ros20] Rosetta Code. *Bitmap/Write a PPM file* [en línia]. 2020. [Consulta: 27/03/2020]. Disponible a: https://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C.

[Net20] Wikipedia. *Netpbm* [en línia]. Wikimedia Foundation, 2020. [Consulta 28/03/2020]. Disponible a: https://en.wikipedia.org/wiki/Netpbm.

[Nor20] Wolfram. *Normalized Vector* [en línia]. Wolfram Research, 2020. [Consulta 28/03/2020]. Disponible a: https://en.wikipedia.org/wiki/Netpbm.

[How16] sbabbi. «How to convert a 3D point on a plane to UV coordinates?» [pregunta]. A: StackOverflow [en línia]. 10 de novembre 2016; 19:04 CET [Consulta: 29/04/2020]. Disponible a: https://stackoverflow.com/questions/18663755/how-to-convert-a-3d-point-on-a-plane-to-uv-coordinates.

[Med08] mrl. *Ray Tracing II* [en línia]. NYU Media Research Lab, 2008. [Consulta 20/08/2020]. Disponible a: https://mrl.nyu.edu/~dzorin/rend05/lecture2.pdf.

[Nor08] mrl. *Normals* [en línia]. NYU Media Research Lab, 2008. [Consulta 20/08/2020]. Disponible a: https://mrl.nyu.edu/~dzorin/intro-graphics-f01/lectures/normals.pdf .

[Int16] Scratchapixel. *Introduction to Ray Tracing: a Simple Method for Creating 3D Images* [en línia]. 2016. [Consulta 14/07/2020]. Disponible a: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell.

[Sha16] Scratchapixel. *Introduction to Shading* [en línia]. 2016. [Consulta 8/10/2020]. Disponible a: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel.

[Min16] Scratchapixel. *A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.)* [en línia]. 2016. [Consulta 7/10/2020]. Disponible a: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/.

[Mat20] Wikipedia. *Matriu identitat* [en línia]. Wikimedia Foundation, 2020. [Consulta 22/07/2020]. Disponible a: https://ca.wikipedia.org/wiki/Matriu_identitat.

[Tra20] Wikipedia. *Translation (geometry)* [en línia]. Wikimedia Foundation, 2020. [Consulta 23/07/2020]. Disponible a: https://en.wikipedia.org/wiki/Translation_(geometry).

[Gen16] Scratchapixel. Ray-Tracing: Generating Camera Rays [en línia]. 2016. [Consulta 8/10/2020]. Disponible a: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/.