UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES I ENGINYERIA INFORMÀTICA

Treball final de grau

ESTRATÈGIES D'OPTIMITZACIÓ DE VISUALITZACIÓ DE POINT CLOUDS

Autor: Víctor González Porras

Director: Dra. Anna Puig
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 14 de gener de 2026

Abstract

This thesis focuses on optimization methods for the visualization of 3D point clouds in real time. These clouds have a lot of points, usually in the order of millions, this makes it so their visualization becomes computationally very costly and hard to do in real time, and so optimization techniques have to be introduced. In this context, this thesis focuses on two methods: the use of compute shaders and the use of the Lebesgue curve for the ordering of the points of the point cloud. In the first part of this thesis it formalizes and studies concepts such as the concept of a point cloud or the Lebesgue curve. It also explores other methods, like the use of traditional graphic pipeline or the elimination of Z-Test, and other orders, like randomly ordering the points or shuffling the points of the Lebesgue curve. The second part of this thesis consists in the design and implementation of a program that incorporates all the studied methods, so that one is able to visualize these point clouds in real time. It also includes a comparison of all the implemented methods and orders, with the objective of finding the best combination of rendering method and ordering of the points depending on the situation.

Resum

Aquest Treball de Final de Grau tracta mètodes d'optimització per a visualitzar núvols de punts (*point clouds*) en 3D en temps real. Aquests núvols acostumen a tenir una quantitat molt gran de punts, de l'ordre de milions, per tant la seva visualització pot esdevenir computacionalment costosa i difícil de fer en temps real, fet que fa necessari introduir tècniques d'optimització. En aquest context, el projecte es centra en dos mètodes: l'ús de *compute shaders* i l'ús de la corba de Lebesgue per ordenar els punts del núvol. En la primera part del projecte es formalitzen i s'aprofundeix en conceptes com el de núvol de punts i de corba de Lebesgue. També s'exploren altres mètodes, com l'ús de la *pipeline* gràfica tradicional o l'eliminació del Z-Test, i altres ordres, com ordenar els punts aleatòriament o barrejar els punts de la corba de Lebesgue. La segona part del TFG consisteix en el disseny i implementació d'un programa que incorpora tots els mètodes estudiats, de tal manera que es puguin visualitzar aquests núvols de punts en temps real. També inclou una comparativa entre tots els mètodes i ordres implementats, per tal de trobar quina és la combinació òptima de mètode de renderitzat i ordre dels punts depenent del cas.

Resumen

Este Trabajo de Fin de Grado trata métodos de optimización para visualizar nubes de puntos (*point clouds*) en 3D en tiempo real. Estas nubes suelen tener una cantidad muy grande de puntos, del orden de millones, por tanto su visualización puede ser muy costosa computacionalmente y difícil de hacer en tiempo real, hecho que hace necesaria la introducción de técnicas de optimización. En este contexto, el proyecto se centra en dos métodos: el uso de *compute shaders* y el uso de la curva de Lebesgue para ordenar els puntos de la nube. En la primera parte del proyecto se formalizan y se profundiza en conceptos como el de nube de puntos y de curva de Lebesgue. También se exploran

otros métodos, com el uso de la *pipeline* gráfica tradicional o la eliminación del Z-Test, y otros órdenes, como ordenar los puntos aleatoriamente o mezclar los puntos de la curva de Lebesgue. La segunda parte del TFG consiste en el diseño y la implementación de un programa que incorpora todos los métodos estudiados, de tal manera que se puedan visualizar estas nubes de puntos en tiempo real. También incluye una comparativa entre todos los métodos y órdenes implementados, para encontrar cuál es la combinación óptima de método de renderizado y orden de los puntos dependiendo del caso.

Agraïments

Vull agrair a la meva tutora, Anna Puig, per la seva ajuda i suport durant tot el transcurs del treball, i a la meva família i amics per estar al meu costat.

Declaració d'autoria

Per la present declaro que s'ha fet ús de ChatGPT en la part del programari, per assessorar alguns scripts de Python i per l'ajut a la generació dels CMakes en relació a l'ús d'algunes dependències entre llibreries. Tot el contingut generat ha estat revisat críticament i validat per l'autor, que assumeix tota la responsabilitat de la versió final del Treball Fi de Grau lliurat (tant del manuscrit com del codi). A més, val a dir que no s'ha utilitzat cap sistema d'IA generativa per generar les idees, els anàlisis o els resultats relatius al present treball.

Índex

1 Introducció **1**

1.1 Context i motivació 1

1.2 Objectiu principal del projecte 2

1.3 Objectius específics 3

1.3.1 Planificació 3

1.3.2 Organització del document 3

2 Antecedents **5**

2.1 Renderització de point clouds 5

2.1.1 Assignació de primitives gràfiques 5

2.1.2 Ray tracing vs Projecció 7

2.1.3 Il·luminació 8

2.2 Càlcul de la Visibilitat i Ordenació Espacial 9

2.3 Optimitzacions de GPU 12

2.4 Càlcul de la visibilitat a nivell de píxel en la GPU 13

2.5 Anàlisi comparativa 14

3 Teoria matemàtica **17**

3.1 Point clouds 17

3.2 Octrees 18

3.3 Càmera 19

3.3.1 Definició de la càmera 19

3.3.2 ViewMatrix 20

3.3.3 Projecció 23
3.4 Space-Filling Curves (SFC) 25
3.4.1 Corbes contínues 26
3.4.2 Corbes discretes 28
3.4.3 Preservació de localitat 31
3.4.4 Relació amb quadtrees i octrees 33
3.4.5 Reordenació respecte a la posició de la càmera 35
3.5 Operacions de Permutació i Shuffle 38
3.5.1 Permutacions 38
3.5.2 Algorismes de shuffle 39
3.5.3 Shuffle Morton 40

4 Desenvolupament del software **42**
4.1 Arquitectura del Sistema 42
4.1.1 Pipeline gràfica 42
4.1.2 Inicialització, configuració i lectura de dades 43
4.2 Implementació del Codi Morton 44
4.3 Càlcul Color 46
4.4 Renderització i Visualització 46
4.4.1 Pipeline clàssica amb GLSL 46
4.4.2 Tècniques d'Optimització en Compute Shaders i GLSL 47

5 Resultats Experimentals **51**
5.1 Hardware utilitzat i Programari desenvolupat 51
5.2 Metodologia 52
5.2.1 Descripció dels datasets 52
5.2.2 Mètriques utilitzades 52
5.2.3 Mètodes avaluats 53
5.3 Resultats 53

6 Conclusions i Treball Futur **58**

6.1 Conclusions 58

6.2 Treball futur 59

A Resultats de la versió de Windows **60**

Capítol 1

Introducció

1.1 Context i motivació

El camp dels Gràfics per Computador (o *Computer Graphics*) ha experimentat un gran creixement en els últims anys, principalment degut als avenços en les unitats de processament gràfic (GPUs),que han ampliat les seves capacitats més enllà del renderitzat tradicional [5] i s'han convertit en una peça clau per a aplicacions de computació en temps real i visió artificial intensiva en dades [23]. Gràcies als gràfics per computador, podem visualitzar tot tipus d'objectes 3D i superfícies, fent d'aquest camp indispensable per a diversos propòsits, des de videojocs fins a mostrar els resultats de recerca científica.

Existeixen diverses formes de representar objectes tridimensionals, entre les quals destaquen les malles poligonals, les representacions volumètriques i els núvols de punts (*point clouds*). Les malles poligonals, usualment formades per triangles, són la representació més estesa, ja que descriuen de forma explícita la geometria i la connectivitat de la superfície, fet que facilita renderitzar-les amb la *pipeline* gràfica tradicional [5]. Per altra banda, les representacions volumètriques utilitzen una discretització de l'espai en vòxels per a descriure l'objecte, proporcionant una descripció completa del volum però amb un cost elevat en memòria i càlcul [35]. En aquesta situació, els núvols de punts representen una alternativa més flexible, ja que són una representació dels objectes que utilitza únicament punts aïllats, sense connectivitat [46]. Cada punt del núvol conté propietats locals de l'objecte, com poden ser la posició, el color o la normal, permetent representar la superfície de forma implícita. Aquesta classe de representació és especialment adequada per a dades obtingudes directament de sensors com escàners làser o LIDARS [21] o càmeres RGB-D (*RedGreenBlue-Depth*) [48], ja que evita processos de reconstrucció geomètrica complexos i preserva la fidelitat de les dades originals.

Els escàners LIDAR emeten rajos a la superfície que es vol escanejar i, mesurant el temps que triga el raig a impactar i tornar, així com l'angle inicial amb que es llença el raig, calculen la posició dels punts. La fotogrametria (o càmeres RGB-D), per altra banda, utilitza un conjunt de fotografies de l'objecte des de diferents angles, i agafant punts coincidents en diverses fotografies es determina la posició del punt en l'espai. Depenent de la tècnica utilitzada i de l'objecte o escena que es vol representar, aquests núvols de punts poden tenir des de un parell de milers fins a bilions de punts.

Encara així, renderitzar o visualitzar núvols de punts, especialment per als núvols de punts més grans, és computacionalment costós, i ho és encara més si volem poder visualitzar-les en temps real amb una bona quantitat de FPS. És per això que sorgeix la necessitat d'optimitzar els algorismes de renderització de núvols de punts el màxim possible, que és el repte adreçat en el treball, tant per la part de computació en si mateixa com de estructures d'organització espaial calculades amb anterioritat, com poden ser els octrees [40], arbres binaris [9, 17], o ordenació dels punts mitjançant corbes de Morton o Hilbert [24, 11].

En quant al procés de renderitzat, en aquest treball s'estudien tècniques implementades en el *pipeline* gràfic convencional incorporat a la majoria de processadors gràfics o GPUs. Concretament, la *graphic pipeline*, com es mostra a la Figura 1.1, descriu les quatre etapes en què es divideix el procés de renderitzar una escena 3D: Aplicació, Processament de geometria, Rasterització i Processament de píxels [5]. La fase d'aplicació acostuma a executar-se a la CPU, encara que es pot involucrar la GPU utilitzant *compute shaders*. Aquí es calculen col·lisions, físiques... A la següent fase es calcula tot el que s'ha de dibuixar i on s'ha de dibuixar amb transformacions i projeccions. La rasterització consisteix a agafar les primitives (triangles, línies, en el nostre cas punts) i trobar tots els píxels que es troben a dins. Finalment, el processament de píxel s'encarrega de decidir quin color tenen aquests píxels i calcular la seva visibilitat. Aquestes tres fases s'executen totes a la GPU.

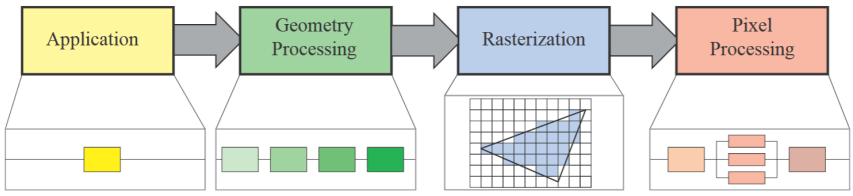


Figura 1.1: La construcció bàsica de la *graphic pipeline*. Cadascuna d'aquestes etapes pot ser una *pipeline* per si mateixa, com es veu a l'etapa de processament de geometria, o una etapa pot estar (parcialment) paral·lelitzada, com a l'etapa de processament de píxels. En aquesta il·lustració, l'etapa d'aplicació és un únic procés, però podria no ser-ho. Imatge extreta de Real Time Rendering [5].

1.2 Objectiu principal del projecte

L'objectiu principal d'aquest treball és formalitzar, implementar i validar diferents mètodes de visualització de núvols de punts i d'ordenació dels punts, amb la finalitat d'analitzar quina és la combinació òptima segons el cas. A més, es vol desenvolupar un programa que implementi tots els mètodes estudiats i que actuï com a marc experimental per tal de poder fer una comparació, tant des del punt de vista visual com en termes d'eficiència temporal.

1.3 Objectius específics

L'objectiu general es desglossa en els següents objectius específics:

- Estudiar i entendre què és un núvol de punts i quina és la seva estructura.
- Investigar i analitzar els diferents mètodes i tècniques de visualització de núvols de punts.
- Establir uns fonaments matemàtics per formular el problema. Això inclourà definir els conceptes de núvol de punts o corba de Morton. També s'explicaran els conceptes de permutació i de la càmera i les seves transformacions i projeccions associades. A més a més, s'inclouran càlculs pel cost dels mètodes presentats.
- Desenvolupament d'un programa capaç de visualitzar núvols de punts amb tots els mètodes que s'estudiaran en aquest treball.
- Validació dels diferents algorismes de renderització de *point clouds* i d'ordenació dels punts dels *point clouds* tant a nivell visual com en termes d'eficiència temporal (utilitzant mètriques com els FPS o frames per segon).

1.3.1 Planificació

Com es pot veure a la figura 1.2, el procés ha sigut planificat durant 4 mesos. Les etapes més llargues han estat les de recerca i codificació. Gràcies a l'assignatura de Gràfics i Visualització de Dades aquest projecte ha sigut molt més fàcil de fer, ja que es van estudiar molts dels conceptes d'aquest treball.

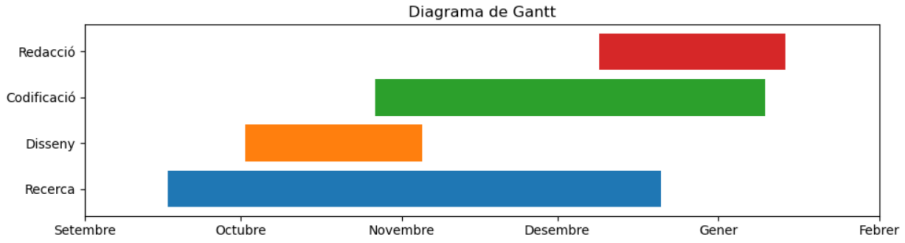


Figura 1.2: Diagrama de Gantt del treball planificat de Setembre a Gener.

1.3.2 Organització del document

L'estructura d'aquesta memòria es desglossa en els següents capítols:

- **Introducció:** El capítol de la Introducció té el propòsit de contextualitzar el projecte esmentant amb breus descripcions l'objectiu principal d'aquest.
- **Antecendents**: Capítol on s'estudien possibles solucions ja fetes als problemes plantejats en el capítol anterior, tant des del punt de vistes de mètodes necessaris com d'aplicacions similars.

- **Teoria Matemàtica:** Capítol on es detalla tota la nomenclatura i la formalització del problema i la solució, amb les propietats rellevants que porten a cada solució.
- **Desenvolupament del software:** En aquest capítol es descriu el desenvolupament realitzat basat en un sistema de *shaders* i es detallen els *compute shaders* realitzats.
- **Resultats Experimentals:** En el capítol de Resultats i Simulacions es mostren els resultats del projecte desenvolupat.
- **Conclusions i treball futur:** En aquest últim capítol es detallen les conclusions i l'estat del projecte de cara al futur.
- **Apèndix:** L'apèndix conté material addicional relacionat amb els resultats del treball.

Capítol 2

Antecedents

En aquest capítol s'introdueix el problema que es vol resoldre i s'analitzen alguns dels mètodes que s'utilitzen a l'hora de renderitzar *point clouds*.

En aquest capítol, els diferents mètodes i tècniques s'han organitzat seguint un ordre lògic que reflecteix el procés de visualització d'un núvol de punts, des de la representació en termes de primitives fins al càlcul de la visibilitat a nivell de píxel i les optimitzacions de la GPU. Més concretament, es presenta una anàlisi estructurada de les diferents fases del procés de visualització de núvols de punts, incloent la representació geomètrica mitjançant primitives, els algorismes de visibilitat i d'il·luminació, les tècniques d'ordenació espaial, les optimitzacions de GPU i la resolució de conflictes de visibilitat a nivell de píxel. L'estudi d'aquests diferents nivells del procés de visualització permet contextualitzar el Treball de Final de Grau.

2.1 Renderització de point clouds

Segons Kivi et al. [29], la recerca sobre com renderitzar *point clouds* va començar a l'any 1985, quan Levoy i Whitted [33] van proposar utilitzar punts com a primitiva per generar imatges 2D d'objectes en 3D. Ells argumentaven que l'augment en la complexitat i la mida de les escenes faria favorable l'ús de punts en comptes de triangles. Abans d'aquest article, la principal primitiva utilitzada per renderitzar eren triangles i altres polígons posicionats en una malla [15].

En aquesta secció, es tracten els tres problemes principals a l'hora de renderitzar *point clouds*, que són: l'assignació d'objectes geomètrics amb volum pels punts, la selecció de l'estratègia de visualització (*ray tracing* o projecció) i el càlcul de la il·luminació.

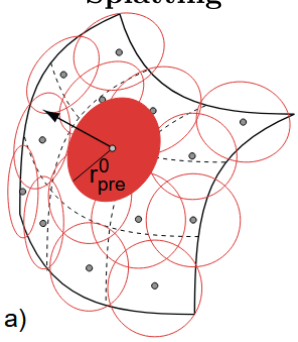
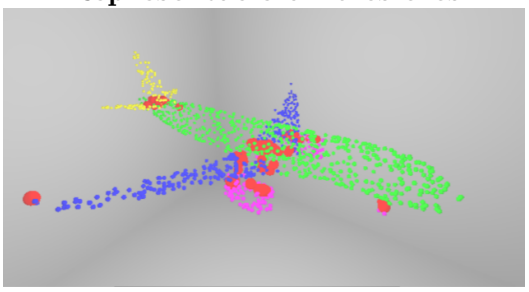
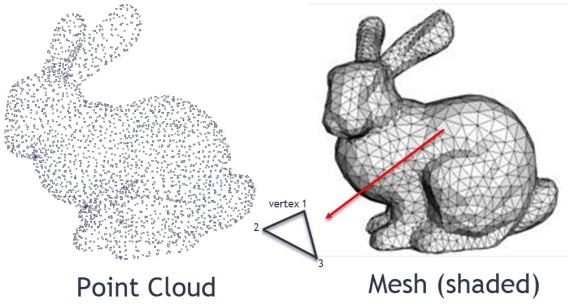
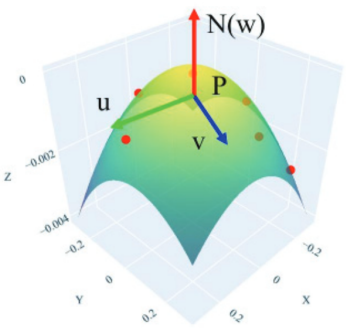
2.1.1 Assignació de primitives gràfiques

Els punts són objectes geomètrics de dimensió 0, és a dir, no tenen longitud ni àrea ni volum, de tal manera que resulta impossible visualitzar-los tal com són. És per això que a la visualització de punts cal assignar una primitiva gràfica amb àrea o volum finits

a cada punt. Aquesta decisió no és trivial, ja que tant la qualitat visual com el cost computacional del renderitzat depenen de la forma i mida de la primitiva escollida.

En aquest context sorgeixen moltes tècniques de representació, entre les quals destaquen quatre de les més comunes: *splatting*, representació amb esferes, conversió a malla i representació amb funció implícita. Aquestes quatre tècniques estan representades a la taula 2.1.

El *splatting* consisteix a representar els punts com discs plans, orientats de forma perpendicular a la normal de cada punt [41]. Aquesta tècnica és àmpliament utilitzada, ja que ofereix un bon equilibri entre simplicitat, eficiència computacional i qualitat visual. La representació amb esferes, com el seu nom indica, consisteix a representar cada punt amb una esfera centrada al punt. És una tècnica més costosa computacionalment que fer *splatting*, però dona una visualització més realista del *point cloud* [31]. D'altra banda, la conversió a malla transforma el núvol de punts en una malla poligonal, habitualment formada per triangles, permetent així l'ús de tècniques de renderització tradicionals basades en la *pipeline* gràfica clàssica [30]. La representació amb funció implícita consisteix a ajustar una funció matemàtica que defineix una superfície contínua a partir del núvol de punts, proporcionant una representació suau i contínua de la forma original, però amb un cost computacional elevat en el procés de reconstrucció [13].

Splatting 	**Representació amb esferes** 
Conversió a malla 	**Funció implícita** 

Taula 2.1: Visualització de les distintes primitives gràfiques estudiades. Imatge de *splatting* extreta de [40]. Imatge de representació amb esferes extreta de [39]. Imatge de conversió a malla extreta de [42]. Imatge de funció implícita extreta de [34].

Tot i l'existència de totes aquestes tècniques, no hi ha cap que sigui universalment òptima per a tots els núvols de punts. Escollir la representació més adequada depèn de

molts factors, com la densitat del núvol, la distribució dels punts o el nivell de detall que es vol aconseguir. En particular, la mida de les primitives per representar els punts té un impacte molt gran a la imatge final, ja que si la mida és massa petita l'objecte pot presentar forats, i si és massa gran els punts es poden solapar i produir artefactes visuals. A més, la representació escollida determina quines estratègies de renderització es poden aplicar i quin és el seu rendiment. Algunes tècniques prioritzen la simplicitat i l'eficiència en els càlculs, mentre que d'altres ofereixen una representació més precisa a canvi de tenir un cost més elevat. Per aquest motiu, trobar un equilibri adequat entre qualitat visual i eficiència és un dels principals reptes en la visualització de núvols de punts.

2.1.2 Ray tracing vs Projecció

Escollir l'estratègia de renderització d'un núvol de punts és un altre qüestió que no és trivial, ja que s'han de tenir en compte aspectes com l'eficiència de cada mètode o la qualitat visual. Existeixen diverses estratègies que es poden utilitzar per renderitzar un núvol de punts, i s'acostumen a dividir segons si iteren sobre els objectes de l'escena (*object order*) o si iteren sobre els píxels (*image order*). Entre totes les estratègies possibles, en destaquen dues: la rasterització (ordre d'objectes) i el *ray tracing* (ordre d'imatge).

Quan es parla de renderització en 3D, el renderitzat basat en projecció o rasterització ha sigut una de les estratègies més utilitzades. La rasterització, il·lustrada a la figura 2.1, consisteix a projectar els punts a la pantalla mitjançant transformacions geomètriques, superposant els que estan més a prop per sobre dels que estan més lluny, utilitzant estructures com el Z-Buffer per a fer comparacions de profunditats. Aquesta forma de visualitzar punts és extremadament eficient, però pot provocar que es vegin forats al *point cloud*, encara que hi ha articles que parlen de com solucionar aquest problema [40]. A la rasterització també sorgeix el problema de que, si el *point cloud* és molt detallat, alguns dels detalls seran més petits que un píxel, i no es podran visualitzar correctament.

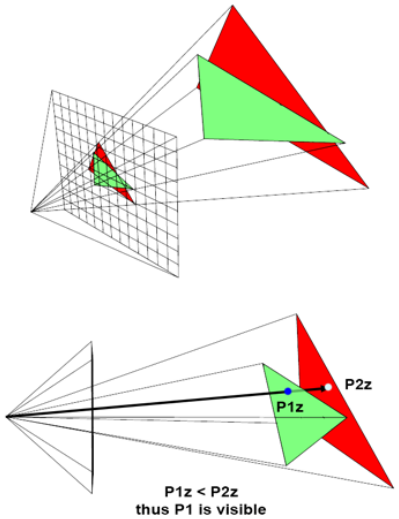


Figura 2.1: Diagrama que mostra el funcionament de la rasterització. Figura extreta de [47].

En quant al *ray tracing*, il·lustrat a la figura 2.2, és una tècnica que consisteix a llençar

raigs de llum a cada píxel de la imatge, seguint el camí que recorren al llarg de l'escena i calculant com canvia la il·luminació cada cop que els raigs col·lisionen amb un objecte de l'escena. Al 1980, Whitted va desenvolupar el que seria una primera versió de *ray tracing* [53]. Aquests algorismes de *ray tracing* són computacionalment molt més costosos que simplement projectar els punts, és per això que visualitzar point clouds en temps real utilitzant *ray tracing* només ha estat possible en anys recents [45]. Quan s'aplica aquest mètode a *point clouds*, s'ha d'assignar alguna figura amb volum als punts d'aquest, ja que els punts no tenen volum i, per tant, seria impossible que els raigs de llum els intersequin. Alternativament, es podria fer que els raigs de llum tinguin volum, fent-los cilindres en comptes de rectes, com ja van fer Schaulfer i Jensen en aquest article [44].

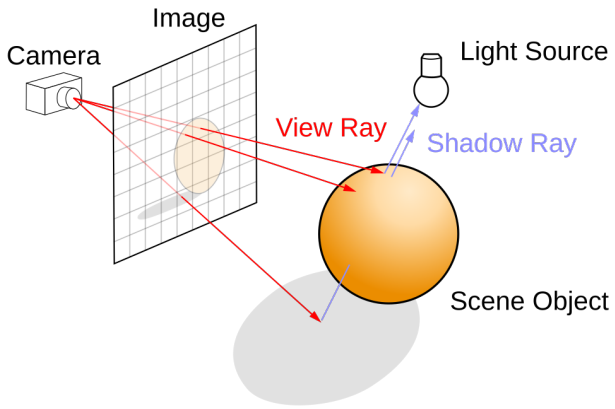


Figura 2.2: Diagrama que mostra el funcionament del *ray tracing*. Figura extreta de [20].

2.1.3 Il·luminació

La il·luminació és un dels elements clau per a poder percebre la forma, la profunditat i el relleu d'escenes tridimensionals. La il·luminació usualment es calcula sobre superfícies definides, ja sigui amb algorismes d'il·luminació local (Lambert o Blinn-Phong) o global (ombres, reflexions o transparències) [5], utilitzant dades locals de la superfície com les normals. Els núvols de punts, en no tenir una representació explícita de la superfície, dificulten aquests càlculs.

Com els punts són elements geomètrics de dimensió 0, no tenen ni àrea ni orientació associats. Per tal de calcular la il·luminació, es necessari aproximar aquestes propietats, normalment usant les normals de cada punt, obtingudes a partir dels veïns locals de cada punt. Aquestes normals permeten utilitzats mètodes similars als usats en malles poligonals, però amb limitacions degut a la natura discreta i desconnectada del núvol de punts.

La representació dels punts, explicada a la secció 2.1.1, té un impacte en el càlcul de la il·luminació i la qualitat visual de la imatge. En tècniques com *splatting*, on cada punt es representa amb un disc orientat per la normal, la il·luminació es pot calcular assumint que hi ha una superfície local plana. Aquesta aproximació és simple i eficient, però pot causar artefactes visuals si la densitat dels punts és massa baixa o les normals no estan ben definides. En el cas de la representació amb esferes, la il·luminació es calcula amb la normal de la superfície de l'esfera, proporcionant una imatge més realista. Aquesta aproximació però incrementa significativament el cost computacional i pot introduir efectes

visuals no desitjats, com ara una percepció excessiva de volum. Altres aproximacions, com la conversió del núvol de punts a una malla poligonal o a una funció implícita, permeten aplicar tècniques d'il·luminació més robustes i coherents, ja que proporcionen una superfície contínua sobre la qual definir normals i propietats materials. Aquests mètodes però requereixen processos previs de reconstrucció geomètrica que poden ser costosos i que s'allunyen de l'objectiu de visualització directa i eficient de núvols de punts en temps real.

En general, la il·luminació en núvols de punts presenta diversos reptes, com ara la dependència de la qualitat de les normals, la presència de soroll en les dades, la variabilitat en la densitat dels punts i la manca de connectivitat explícita. Aquests factors poden provocar artefactes visuals, inconsistències en la il·luminació i dificultats per percebre correctament la forma de l'objecte. Per aquest motiu, l'elecció de la primitiva gràfica i de l'estratègia de renderització resulta fonamental per assolir un equilibri adequat entre qualitat visual i eficiència computacional. En aquest treball però, la il·luminació no constitueix l'objecte principal d'estudi, sinó que s'utilitza únicament com a eina per facilitar la percepció visual del núvol de punts.

2.2 Càlcul de la Visibilitat i Ordenació Espacial

Un dels principals reptes en la visualització de núvols de punts és el càlcul eficient de la visibilitat dels punts, especialment quan es treballa amb conjunts de punts molt grans.

En aquest cas, el problema de la visibilitat està relacionat amb la projecció dels punts sobre la pantalla. Quan diversos punts es projecten sobre el mateix píxel, només el punt més proper a la càmera contribueix a la imatge final, mentre que la resta queden ocults. Això implica que una part significativa del treball de processament es dedica a punts que no tenen cap impacte visual, fet que resulta especialment ineficient en núvols de punts densos.

Aquest problema depèn de la distància de la càmera a l'escena: quan la càmera està a prop, cal una alta densitat de punts per representar correctament els detalls de la superfície, mentre que quan la càmera s'allunya, la major part dels punts es projecten sobre un nombre reduït de píxels. En aquest darrer cas, processar tots els punts resulta ineficient, ja que es processen punts que no apareixen a la imatge final. Per solucionar aquests problemes, s'han desenvolupat diverses estratègies basades en l'ordenació espacial de les dades, que permeten seleccionar dinàmicament quins punts són rellevants en funció de la vista i establir diferents nivells de detall (*Level of Detail* o LOD). Aquestes tècniques tenen com a objectiu reduir el nombre de punts processats i renderitzats sense comprometre significativament la qualitat visual de la imatge final. Això es fa normalment utilitzant estructures d'ordenació espacial que permeten seleccionar subconjunts del núvol de punts segons la distància a la càmera o segons la mida projectada en pantalla. D'aquesta manera, quan la càmera està lluny, s'utilitza una representació menys densa que evita projectar múltiples punts sobre els mateixos píxels, mentre que en vistes properes s'incrementa la densitat per preservar el detall geomètric. Aquest enfocament permet reduir significativament el nombre de punts processats i, per tant, el cost del càlcul de la visibilitat, mantenint alhora una qualitat visual coherent amb la resolució de la imatge.

Els primers mètodes en utilitzar estructures d'aquest tipus van ser Surfels [40] i QSplat [43] a l'any 2000. Surfels utilitza una estructura de dades basada en quadtrees i octrees, que són un tipus d'arbre on cada node té 4 i 8 fills respectivament. Aquesta estructura es mostra a la figura 2.3 en el cas 2D, on es poden veure dos nivells de l'arbre i la divisió de l'espai que formen. D'altra banda, QSplat es basa en una jerarquia de *bounding spheres*, és a dir, una esfera que engloba tots els punts que seria l'arrel de l'arbre amb esferes més petites com a fills. A la figura 2.4, es mostra l'estructura de l'arbre usat per la jerarquia, com es guarda aquest arbre i les propietats de cada node. Dachsbacher et al. [9] van proposar els *Sequential Point Trees* (SPT), un arbre similar a l'utilitzat a QSplat i que després es passa a una taula de forma seqüencial, ordenada per la jerarquia de l'arbre original. Aquesta estructura és més òptima per a la GPU, ja que recórrer una taula és més simple que recórrer un arbre.

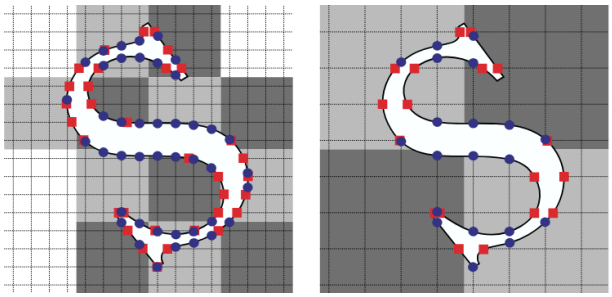


Figura 2.3: Divisió de l'espai 2D d'un quadtree. La figura de la dreta representa el primer nivell del quadtree, que divideix l'espai en quatre. La figura de l'esquerra és el segon nivell, i es veuen les 16 divisions. Els quadrats vermells i els cercles blaus representen diferents nivells dels *surfels* o elements de superfície. Imatge extreta de l'article de Surfels [40].

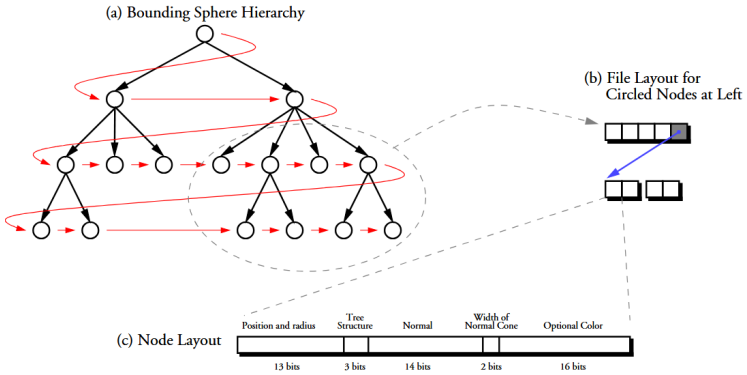
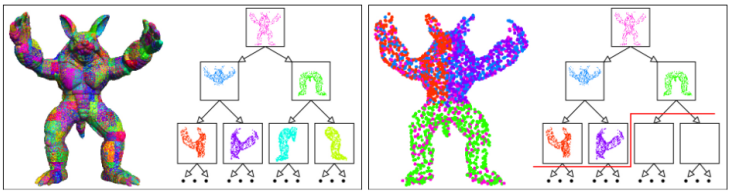


Figura 2.4: Estructura de dades de QSplat. (a) L'arbre es guarda en BFS (*Breadth First Search*), marcat per les fletxes vermelles. (b) L'enllaç entre els nodes i els seus fills s'estableix per mitjà d'un punter. (c) Cada node té 48 bits (32 sense color). Imatge extreta de l'article de QSplat [43].

Per una altra banda, Gobbetti i Marton [17] van crear els *Layered Point Clouds* (LPC), un tipus d'arbre binari que divideix l'espai en dos a cada nivell. Aquesta estructura es mostra a la figura 2.5, on es pot veure com es divideixen els punts i com es pot reconstruir una imatge amb menys detall. Al contrari que en QSplat, en aquest mètode els nodes de l'arbre no són esferes, sinó *point clouds* més petits. A més a més, a cada node no hi són tots els punts que corresponen a la seva divisió de l'espai, només un subconjunt de punts,

que és disjunt al conjunt de punts del node pare, és a dir, no comparteixen punts. En renderitzar, inicialment només es mostren els punts del node arrel, i després, depenent de la distància de la càmera, es van afegint els punts dels nodes fills per fer el LOD, cosa que és molt més ràpida de calcular i s'ha convertit en un estàndard per visualitzar *point clouds* grans.



(a) Full structure (b) Adaptive resolution

Figura 2.5: Estructura dels *Layered Point Clouds*. El model multiresolució té exactament el mateixos punts del model inicial, però agrupats en grups de mida constant i organitzats en una estructura LOD. Representacions de resolució variable es poden obtenir definint un tall de la jerarquia i unint tots els nodes per sobre del tall. Imatge extreta de l'article de Gobbetti i Marton [17].

Fins ara s'han descrit estructures d'ordenació espacial explícites, com ara arbres jeràrquics, que permeten organitzar els punts segons la seva posició a l'espai. A més, existeixen alternatives que no realitzen la construcció d'una estructura de dades explícita, sinó que es basen únicament en l'ordre en què s'emmagatzemen els punts. En aquests casos, la mateixa seqüència de punts defineix implícitament una organització espacial, sense necessitat de nodes ni jerarquies addicionals. Aquest tipus d'ordenació es basa en les anomenades *Space-Filling Curves*, que són funcions que recorren l'espai multidimensional seguint un camí continu i el transformen en una seqüència unidimensional [6]. L'objectiu principal d'aquestes corbes és preservar la localitat espacial, és a dir, aconseguir que punts pròxims a l'espai també apareguin propers en la seqüència ordenada. Bader [6] descriu la construcció i les propietats de diverses d'aquestes corbes, entre les quals destaquen les corbes de Morton i de Hilbert, així com la manera com codifiquen les coordenades espacials.

La corba de Hilbert és una corba d'ompliment de l'espai que recorre l'espai de manera contínua i recursiva, visitant totes les regions sense salts bruscos. El seu disseny permet preservar millor la localitat espacial, de manera que punts propers a l'espai també tendeixen a ser propers en l'ordre lineal.

La corba de Morton, també coneguda com a Z-order, és una manera d'ordenar punts de l'espai convertint les seves coordenades en un sol valor numèric. Això es fa intercalant els bits de les coordenades (per exemple, en 2D, els bits de x i y), fet que produeix un recorregut de l'espai amb una forma característica de Z. Aquestes corbes, les de Hilbert i les de Morton, s'analitzen amb mes profunditat al capítol 3.

Aquestes ordenacions són especialment interessants per repartir la feina de manera eficient en la visualització de núvols de punts usant la GPU [26]. El problema principal és que no totes les parts d'una escena requereixen la mateixa quantitat de càlculs: hi ha zones amb molts punts visibles i d'altres amb molt pocs. Si aquesta feina es reparteix malament, alguns processadors de la GPU poden acabar ràpidament la seva feina mentre d'altres continuen treballant, fet que provoca un ús ineficient del paral·lelisme. En lloc

d'ajustar la distribució de la feina durant l'execució (distribució dinàmica), Kerbl et al. [26] proposen utilitzar patrons fixes o estàtics per repartir el treball. Per fer-ho, ordenen les dades segons un recorregut de l'espai que manté juntes les regions properes, com ara les corbes de Morton i de Hilbert, i assignen blocs consecutius d'aquest ordre als nuclis de la GPU. D'aquesta manera, cada nucli tendeix a processar zones de l'escena amb una càrrega similar.

Els resultats mostren que la corba de Hilbert, en agrupar millor les regions veïnes, aconsegueix repartir la càrrega de manera més equilibrada que la corba de Morton, permetent així un ús més uniforme de la GPU sense necessitat de mecanismes complexos de redistribució dinàmica.

Tot i així, la corba de Morton continua sent una alternativa molt rellevant donada la seva simplicitat computacional: la corba de Morton es construeix mitjançant la intercalació directa dels bits de les coordenades espacials, fet que permet calcular l'ordre dels punts de manera eficient i fàcilment paral·lelitzable. A més, tot i que la preservació de localitat de Morton és inferior a la de Hilbert, en molts casos pràctics ofereix un comportament suficientment bo per reduir l'accés dispers a memòria i millorar la coherència espacial dels punts processats consecutivament. Per aquests motius, la corba de Morton representa un bon compromís entre eficiència computacional i qualitat de l'ordenació espacial, fet que justifica el seu estudi i comparació amb altres mètodes dins d'aquest treball.

2.3 Optimitzacions de GPU

Com s'ha anat mostrant al llarg d'aquest capítol, la visualització eficient de núvols de punts requereix reduir tant el nombre d'operacions innecessàries com el cost associat al processament de dades que no contribueixen a la imatge final. Aquest objectiu no només depèn de decisions a nivell d'aplicació, com l'ordenació dels punts o l'ús de nivells de detall, sinó també de com aquestes dades són processades internament per la GPU. Per aquest motiu, en aquest apartat s'introdueixen alguns conceptes clau relacionats amb l'arquitectura de la GPU que permeten entendre millor com certes decisions de disseny poden afectar directament el rendiment.

En una GPU, la unitat més petita que es pot programar és un *thread*. Els *threads* s'agrupen en conjunts per formar *warps* (NVIDIA), wavefronts (AMD) o subgrups (OpenGL). Aquests *warps* són grups de 32 *threads* que s'executen concurrentment i operen en la base SIMT (Single Instruction Multiple Threads). Normalment, tots els *threads* d'un *warp* executen la mateixa instrucció alhora, però de vegades poden divergir. Quan els *threads* divergeixen en un mateix *warp*, fan camins diferents d'execució (per exemple entren en camins diferents d'un if), fet que provoca que el *warp* deixi de treballar de manera eficient perquè aquestes branques s'han d'executar de forma seqüencial (és a dir es serialitzen). Aquesta característica de l'execució en GPU és especialment rellevant en la visualització de núvols de punts, ja que l'ordre en què es processen els punts pot influir directament en el grau de divergència entre *threads*.

En aquest context, diversos treballs han proposat tècniques per millorar l'eficiència dels *warps* en situacions amb divergència. Khorasani et al.[27] van introduir al 2015 una tècnica anomenada *Collaborative Context Collection* per a augmentar l'eficiència dels

warps quan hi havia divergència, proposant una tècnica en què els *threads* d'un mateix *warp* comparteixen informació sobre el camí d'execució abans de processar els punts, reduint l'impacte de la divergència. Aquest enfocament és molt efectiu quan els punts estan ordenats de manera que els *threads* d'un mateix *warp* processen regions espacials similars. En quant a la utilització de *warps* per visualitzar *point clouds*, Schütz et al.[46] van aprofitar que els *threads* dins d'un *warp* es poden comunicar per a compartir resultats intermedis per reduir el nombre de vegades que s'executaven instruccions costoses. Aquest enfocament és especialment beneficiós quan els punts estan ordenats de manera coherent, ja que els *threads* tendeixen a processar punts amb característiques similars.

A més de l'execució dels *threads*, l'accés a memòria juga un paper fonamental en el rendiment de la GPU. El *memory coalescing* és una tècnica de hardware utilitzada per millorar la utilització de l'ample de banda de la memòria de la GPU, i consisteix en agrupar múltiples lectures lògiques de memòria contigües en un únic accés a memòria. Tot i que aquesta tècnica està implementada de forma transparent en les GPUs modernes, l'organització de les dades a memòria pot facilitar o dificultar aquest procés. Diversos estudis busquen optimitzar la memòria encara més [12] i segons un estudi del 2017 [28], el *memory coalescing* millora el rendiment fins a 2,3 vegades respecte els accessos no coalescents.

2.4 Càlcul de la visibilitat a nivell de píxel en la GPU

En la visualització de núvols de punts, un dels principals reptes, tal i com s'ha comentat a la secció 2.2, és la gestió eficient de la visibilitat quan múltiples punts es projecten sobre el mateix píxel. Les primitives gràfiques tradicionals, com *GL_Points* d'OpenGL, ofereixen una solució on cada punt del *point cloud* es visualitza com un *quad* orientat a la càmera i es delega el problema de la visibilitat completament a la *pipeline* gràfica sense proporcionar un control fi sobre el procés de resolució de conflictes entre *threads*.

En aquest context, diversos treballs han explorat l'ús de *compute shaders* com a alternativa per implementar mètodes de renderització més flexibles. Günther et al.[18] van ser els primers a suggerir utilitzar un *compute shader* per renderitzar punts en comptes de la primitiva *GL_Points* d'OpenGL. La seva implementació a OpenCL esperava fins que un píxel es pogués escriure, el bloquejava, actualitzava el color i la profunditat i el tornava a desbloquejar. Els autors van veure que si s'afegia un Z-Test el *compute shader* escalava molt millor que *GL_Points* amb el nombre de punts per píxel. Posteriorment, Lukač et al.[36] van utilitzar un algorisme basat en una funció `atomicMin` per renderitzar punts. En aquest enfocament, quan diversos punts van a parar al mateix píxel, cada *thread* intenta actualitzar la profunditat mitjançant una operació atòmica, garantint que només es conserva el valor corresponent al punt més proper a la càmera. Aquesta solució permet un control explícit de la profunditat sense necessitat de bloquejos, però introdueix un cost elevat quan molts *threads* accedeixen simultàniament al mateix píxel. En núvols de punts molt densos, aquest fet fa que el rendiment decreixi i apareguin artefactes visuals.

Schütz et al.[46] van estudiar i implementar diverses tècniques de renderitzat amb *compute shaders*: *GL_Points* (per tenir un *shader* base amb què comparar), només amb `atomicMin`, posant Z-Test, reducció a crides d'`atomicMin` a nivell de warp, els dos anteriors junts, entre d'altres. A través d'experiments amb núvols de punts de diferents mides,

van concloure que el millor era un *shader* que incloïa deduplicació a nivell de *warp*, és a dir, que a cada grup SIMT només cridava `atomicMin` un cop, seguit del *shader* que combinava Z-Test amb reducció de crides a `atomicMin` per a les GPUs que no tenen la funció de deduplicar. És a dir, les millors prestacions s'obtenen quan es limita el nombre de crides a `atomicMin`, ja sigui mitjançant deduplicació a nivell SIMT o combinant-les amb mecanismes de control de profunditat.

En resum, tots aquests treballs mostren que existeixen molts paràmetres i estratègies que afecten la resolució del mateix problema fonamental: la resolució eficient de la visibilitat quan múltiples punts projecten sobre el mateix píxel. L'elecció entre primitives gràfiques tradicionals i mètodes basats en *compute shaders*, així com la manera de gestionar els conflictes d'escriptura a nivell de píxel tindran un impacte directe tant en el rendiment com en la qualitat visual del resultat final.

2.5 Anàlisi comparativa

En aquest apartat, per acabar de donar contest a aquest treball, es sintetitzen els principals mètodes explicats i utilitzats per treballs previs. La taula 2.2 resumeix els diferents nivells de decisió involucrats en la visualització de núvols de punts, així com les tècniques estudiades en cadascun d'ells, des de la representació geomètrica dels punts fins a la resolució final de la visibilitat a nivell de píxel i les optimitzacions específiques de GPU. És a dir, la visualització eficient de núvols de punts no depèn d'una única tècnica, sinó de la combinació coherent de decisions a diferents nivells del procés de renderització, que és precisament l'enfocament adoptat en aquest treball.

Aspecte	**Opcions estudiades**	**Objectiu principal**	**Rendiment/Qualitat**
Primitiva gràfica	GL Points (quad), splatting (disc), esfera, funció implícita / malla	Definir com es representa cada punt	Determina el cost per punt i la qualitat visual bàsica
Càlcul de la visibilitat (global)	Z-buffer (rasterització), ray tracing	Decidir què és visible a l'escena	Z-buffer: eficient; ray tracing: més exacte però costós
Il·luminació	Il·luminació local, il·luminació global	Aportar informació visual de forma i relleu	Millora la percepció, però incrementa el cost computacional
Ordenació espacial	LOD, Morton (Z-order), Hilbert	Reduir i organitzar els punts processats	Millora l'escalabilitat i la coherència espacial
Optimitzacions de paral·lelisme	Warps / SIMT, reducció de divergència, memory coalescing, deduplicació	Executar el renderitzat eficientment a GPU	Augmenta l'ocupació de la GPU i redueix costos ocults
Càlcul de visibilitat a nivell de píxel	Z-test, `atomicMin`, deduplicació a nivell de warp	Resoldre conflictes quan molts punts cauen al mateix píxel	Clau per al rendiment en núvols densos

Taula 2.2: Resum dels aspectes de la visualització de núvols de punts estudiats.

A la taula 2.3, es pot trobar una comparació de diferents mètodes utilitzats per a visualitzar núvols de punts. A continuació s'explica en què consisteixen aquests mètodes.

El primer mètode està explicat a l'article de Schütz et al. [46]. En aquest article, els autors estudien diversos mètodes, combinant Z-buffer, operacions atòmiques i deduplicació a nivell de *warp*, però a la taula en destaquem només un, basat en *compute shaders*. El *compute shader* codifica el color i la profunditat de cada punt creant un int de 64 bits, on el color (RGB) ocupa els 24 bits menys significatius i la profunditat els següents 32 bits. D'aquesta manera, si es comparen dos ints, el que és petit té la profunditat més baixa. Després es guarden tots aquests ints en un *framebuffer*, que es converteix en una textura d'OpenGL i es dibuixa la textura a pantalla.

El següent mètode a comparar prové d'aquest article de Schaufler i Jensen [44]. Aquest mètode implementa una versió de *ray tracing* que no utilitza primitives gràfiques amb volum, sinó que es llencen raigs cilíndrics per a detectar col·lisions. També incorpora octrees per organitzar els punts del núvol.

El tercer mètode està extret de l'article de Kashyap et al. [25]. Els autors utilitzen una versió de *ray tracing* que implementa *splatting* per a representar els punts, juntament amb un octree per tenir una jerarquia LOD.

L'últim mètode és l'utilitzat pel Gerard Perelló al seu TFG [39]. Aquí el Gerard fa servir l'estratègia de *ray tracing*, utilitzant esferes per a representar els punts del núvol. També fa servir octrees per a la jerarquia de punts, i implementa una nova estructura que anomena *MultiOctree*, que agrupa múltiples octrees en una única estructura de dades.

Article	Representació dels punts	Estratègia	Il·luminació	Efectes realistes	Optim.	Compute shader
[46]	MipMap de punts	ZBuffer	basada en color i profunditat	No	GPU-based, Morton	Sí
[44]	Punts	Raytracing, raigs en forma de cilindre	Basada en color	Ombres	Octree	No
[25]	Splatting	Raytracing	Basada en color	Ombres	Octree, GPU-based	No
[39]	Esferes	Raytracing	Basada en color	Ombres	Octree, Multi-Octree	No

Taula 2.3: Comparació de mètodes de visualització de núvols de punts a l'estat de l'art.

En el present Treball de Final de Grau, es pren com a punt de partida l'article de Schütz et al. [46], adoptant les idees principals relatives a la gestió de la visibilitat i les optimitzacions a nivell de GPU. A partir d'aquesta base, el treball se centra específicament en l'estudi l'estratègia d'ordenació dels punts usant Morton i en l'anàlisi del seu impacte tant en el rendiment com en la qualitat visual del resultat.

En comparació amb el treball [39], que optimitza el rendiment del *ray tracing* mit-

jançant estructures jeràrquiques complexes com el *multi-octree*, el present Treball de Final de Grau adopta un enfocament diferent orientat al temps real, basat en rasterització, Z-buffer i ordenació espacial dels punts, aconseguint una millor escalabilitat quan l'objectiu és la visualització interactiva de grans núvols de punts.

Capítol 3

Teoria matemàtica

En aquest capítol s'introdueixen definicions bàsiques dels conceptes clau del treball, com són *point clouds*, octrees o la càmera, per poder definir el problema de la necessitat de recorregut en ordre dels punts, que és el que s'ha dit com a conclusió del capítol anterior. Per aquest motiu s'estudiaran les *Space-Filling Curves* (concretament, les corbes de Hilbert i Morton) que presenten propietats ideals per optimitzar la visualització dels *point clouds*. Per això, es vol comprovar que aquestes corbes omplen del tot l'espai (introduint el concepte de la mesura de Lebesgue) i també es vol mesurar les *Space-Filling Curves* en termes de continuïtat, localitat i eficiència. En termes d'eficiència, en paral·lelitzar, sorgeixen problemes de patrons regulars que provoquen que es projectin molts punts de la corba en la mateixa regió de l'espai, fet que provoca divergència i desequilibris de càrrega entre processadors. Per solucionar aquesta regularitat, s'estudien també els mètodes de permutació i *shuffle*.

3.1 Point clouds

Tradicionalment, la representació de superfícies en l'espai tridimensional ha estat feta mitjançant representacions explícites com ara *meshes* poligonals. Tanmateix, hi ha altres opcions, com la representació implícita de superfícies utilitzant *point clouds* [33]. Aquests *point clouds* són conjunts de l'ordre de milions de punts distribuïts per la superfície implícita.

Definició 3.1.1. *Un **point cloud**, o núvol de punts, **P** és una tupla (P,A), on:*

- *P és un conjunt finit de punts* $P = \{p_1, p_2, ..., p_m\} \subset \mathbb{R}^3, m \in \mathbb{N}$.
- *A és l'espai d'atributs associats a cada punt* p_i *de P.*

Notació 1. *Durant la resta d'aquest treball s'utilitzarà el nom "point cloud"per referir-se tant a la tupla* (P, A) *com només al conjunt de punts P.*

A l'espai d'atributs A es troba tota la informació no geomètrica del *point cloud*: el color RGB de cada punt, les seves normals, material... Es pot trobar, però, *point clouds*

que només tenen punts, sense cap atribut. En aquest cas el conjunt A és el buit, i els punts només tenen la seva posició com a informació.

Per poder lligar cada punt de P amb els seus atributs de A, es defineix una **funció d'atribut**, que servirà per assignar a cada punt els seus atributs. En un cas real aquesta funció no existiria i els atributs ja estarien assignats a cada punt, ja sigui mitjançant punters als atributs o representant el punt com una tupla $(x, y, z, r, g, b, ...)$.

Definició 3.1.2. *Sigui $\boldsymbol{P} = (P, A)$ un point cloud amb $A = \{a_1, ..., a_n\}$ atributs diferents. Per a cada atribut a_i definim la seva **funció d'atribut** f_{a_i} com:*

$$f_{a_i} : P \longrightarrow A_{a_i},\ p \longmapsto f_{a_i}(p) \tag{3.1.1}$$

On A_{a_i} es determina en funció de l'atribut a_i (per exemple, $\mathbb{R}^3$ en cas de color RGB).

3.2 Octrees

Un octree, il·lustrat a la figura 3.1, és una estructura de dades basada en arbres utilitzada per dividir l'espai tridimensional en regions més petites. És especialment eficient per representar i organitzar dades 3D de forma jeràrquica. Abans de definir formalment aquest concepte, però, hem de definir el concepte de capsa contenidora d'un *point cloud*.

Definició 3.2.1. *Sigui $\boldsymbol{P}$ un núvol de punts tal com s'ha definit a 3.1.1, la **capsa contenidora** de $\boldsymbol{P}$ és la capsa B que té com a vèrtexs oposats $v_{min} \in \mathbb{R}^3$ i $v_{max} \in \mathbb{R}^3$, on:*

$$v_{min} = (\min_x(p), \min_y(p), \min_z(p)), p \in \boldsymbol{P} \tag{3.2.1}$$

$$v_{max} = (\max_x(p), \max_y(p), \max_z(p)), p \in \boldsymbol{P} \tag{3.2.2}$$

*El **centre** $c \in \mathbb{R}^3$ de la capsa és:*

$$c = v_{min} + \frac{v_{max} - v_{min}}{2} = \frac{v_{max} + v_{min}}{2} \tag{3.2.3}$$

Definició 3.2.2. *Un **octree $\boldsymbol{O}$** basat en un point cloud $\boldsymbol{P}$ és un graf acíclic dirigit definit com una tupla $(\boldsymbol{P}, V, E, v_0, C(v_i), \mathcal{B})$, on:*

- *V és el conjunt de nodes, on cada node $v \in V$ té una capsa contenidora $B(v) \in \mathcal{B}$ i el conjunt de punts $P(v) = \{p_i\}$ del point cloud $\boldsymbol{P}$ continguts dins de la capsa, és a dir, $\forall p_i \in \boldsymbol{P} : p_i \in B(v)$.*
- *E és el conjunt de vèrtexs dirigits que connecten els nodes de V, on $e_{ij} \in E$ connecta el node v_i amb el v_j. Un node v_i està connectat al seu fill v_j si i només si compleix el criteri de divisió $C(v_i)$.*
- *$C(v_i)$ és el criteri de subdivisió aplicat a tots els nodes v_i de l'octree.*

- v_0 *és el node arrel de l'octree, per tant* $v_0 \in V$*, és a dir,* $\forall v_i \in P : \nexists e_{i0}$ *que compleix* $C(v_i)$ *en relació amb* v_0*.*
- $\mathcal{B}$ *és l'espai de capses contenidores, que representen l'espai ocupat per cada node.*

Els nodes de V *poden ser* ***interns*** *o* ***fulles****. Els nodes interns tenen* ***vuit*** *potencials fills amb capses contenidores més petites. La unió de les capses contenidores dels fills és igual a la capsa del node. Els nodes fulla no tenen cap fill. La* ***profunditat*** *d'un node és el nombre de nodes pare que es necessita per arribar al node arrel.*

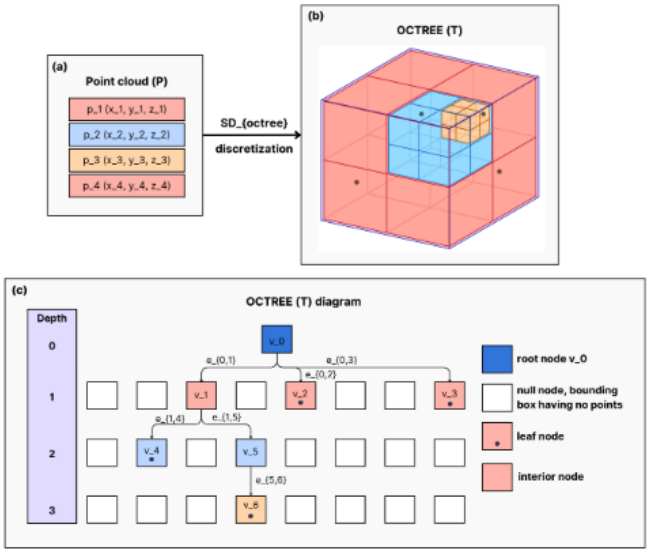


Figura 3.1: Estructura d'un octree, es pot veure tant la representació a l'espai com el diagrama de l'arbre. Figura extreta del TFG del Gerard Perelló [39].

3.3 Càmera

3.3.1 Definició de la càmera

Un dels objectes més importants dins del camp dels gràfics per ordinador és la càmera. És a través de la càmera que es veu l'escena i es pot ser capaç de moure's per aquesta. És molt important saber on està la càmera i cap a on mira en tot moment, i es per això que la càmera té el seu propi sistema de vectors i coordenades, que s'hauran de definir i estudiar per a poder entendre com es relaciona amb les coordenades de món. Aquest sistema de coordenades, que defineix **l'espai de càmera**, necessita un punt que sigui l'origen i tres vectors linealment independents. Els components de la càmera que defineixen aquestes coordenades són:

- **Posició de la càmera (lookFrom)**: és un vector $L = (L_x, L_y, L_z)$ en coordenades de món que apunta a la càmera. Aquest serà el nostre punt d'origen per l'espai de càmera.

- **Punt de referència (VRP)**: És el punt $D = (D_x, D_y, D_z)$ a on la càmera està mirant a cada moment. A OpenGL, la càmera mira cap a l'eix z negatiu.
- **Vector vertical (VUP)**: és un vector $\vec{V} = (V_x, V_y, V_z)$ que defineix la direcció vertical de l'observador. És per tant perpendicular a VRP.

El tercer vector de la base es pot prendre com el producte vectorial de VUP i VRP, per tant, no és un component de la càmera.

Tanmateix, aquests no són els únics components de la càmera. Un cop es tenen definides les coordenades de la càmera, s'ha de definir l'espai que serà mostrat per pantalla, utilitzant el **frust**, il·lustrat a la figura 3.2.

Definició 3.3.1. *El **frust** és una piràmide rectangular truncada de sis cares amb les següents característiques:*

- *Un **pla proper**, perpendicular a VRP, que talla la piràmide a una distància d de la càmera.*
- *Un **pla llunyà**, perpendicular a VRP, a distància f, que és la base de la piràmide. Els punts a distància més petita que d o més gran que f són invisibles.*
- *Un **pla de projecció**, a distància g, també perpendicular a la direcció de la càmera. Normalment $g = f$, per tant, seria el pla llunyà.*
- *La forma rectangular del frust ve determinada per la relació d'aspecte $s = \frac{Viewport_{amplada}}{Viewport_{alcada}}$.*

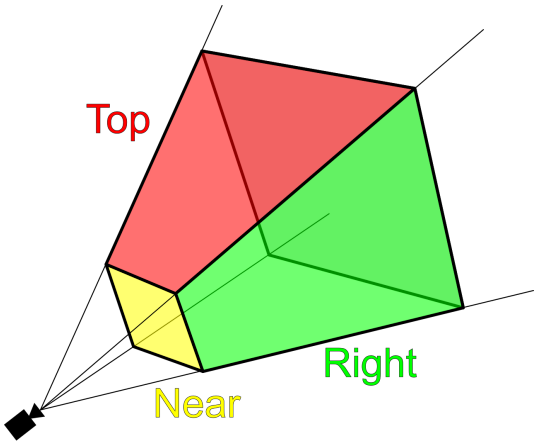


Figura 3.2: Imatge del frust d'una càmera. Imatge extreta de [38]

.

3.3.2 ViewMatrix

La **viewMatrix** és una matriu que guarda les transformacions necessàries per fer el canvi de coordenades de món a coordenades de càmera. És una composició de rotacions i translacions, per tant, en primer lloc es definiran què són les rotacions.

Rotacions

Segons Weisstein[51], una rotació és una transformació ortogonal que preserva l'orientació, que consisteix en girar un objecte o sistema de coordenades un determinat angle al voltant d'un punt fix. Les rotacions es poden expressar en forma de matriu[52]:

Definició 3.3.2. *Es denota per* $M : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^{3\times3}$ *a la funció que, donat una direcció* $\vec{v} \in \mathbb{R}^3$ *i un angle* $\theta \in \mathbb{R}$, $M_{\vec{v}}(\theta)$ *és una* ***matriu de rotació*** *tal que,* $\forall p \in \mathbb{R}^3$, $p' = M_{\vec{v}}(\theta)\cdot p$ *és el resultat de rotar el punt* p *en sentit antihorari* θ *graus al voltant de l'eix definit per* $\vec{v}$.

A $\mathbb{R}^3$, les rotacions respecte dels eixos x, y i z en sentit antihorari quan es mira cap a l'origen estan identificades amb les següents matrius:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{3.3.1}$$

$$R_y(\beta) = \begin{pmatrix} cos(\beta) & 0 & -sin(\beta) \\ 0 & 1 & 0 \\ sin(\beta) & 0 & cos(\beta) \end{pmatrix} \tag{3.3.2}$$

$$R_z(\gamma) = \begin{pmatrix} cos(\gamma) & sin(\gamma) & 0 \\ -sin(\gamma) & cos(\gamma) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3.3}$$

A continuació s'enuncia el teorema de les rotacions d'Euler[50], que permet escriure qualsevol rotació com a combinació de les tres matrius anteriors:

Teorema 3.3.3. ***Teorema de les rotacions d'Euler****: Qualsevol rotació arbitrària es pot descriure amb només tres angles* (α, β, γ).

Els tres angles (α, β, γ) s'anomenen **angles d'Euler** i la matriu de la rotació arbitrària assignada a aquests angles serà el producte de les matrius de cada eix. Si es denota la matriu final com

$$R_{xyz}(\alpha, \beta, \gamma) = R_x(\alpha)R_y(\beta)R_z(\gamma) \tag{3.3.4}$$

Les rotacions es fan d'esquerra a dreta, en aquest cas començant per la rotació amb eix x i acabant per la rotació d'eix z.

Per a la càmera del projecte, s'utilitzen **angles de Tait-Bryan**[1], un tipus de representació d'angles en què s'utilitzen els tres eixos de l'espai. Els tres angles utilizats en aquesta representació es diuen *pitch*, que representa la rotació sobre l'eix z, *yaw*, que representa la rotació sobre l'eix y, i *roll*, que representa la rotació sobre l'eix x. Un exemple d'aquesta representació es troba a la figura 3.3.

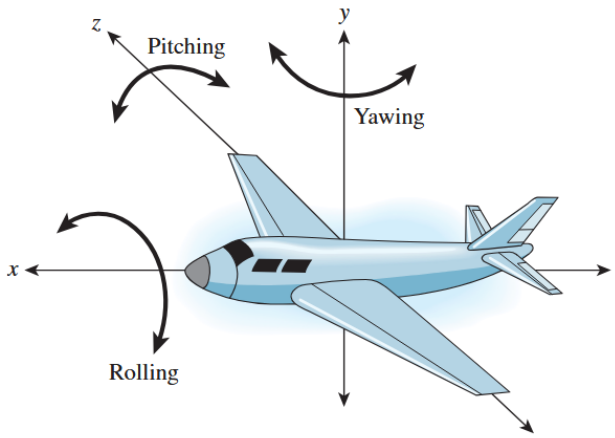


Figura 3.3: Representació dels angles de Tait-Bryan. Imatge extreta de [22]

.

Definició de la viewMatrix

Com ja s'ha dit anteriorment, la **viewMatrix** és una matriu que guarda les transformacions necessàries per fer el canvi de coordenades de món a coordenades de càmera. A continuació es defineix la **viewMatrix**:

Definició 3.3.4. *Sigui $L = (L_x, L_y, L_z)$ la posició de la càmera o lookFrom, $\vec{V} = (V_x, V_y, V_z)$ el vector vertical i $D = (D_x, D_y, D_z)$ el punt a què apunta la càmera. Llavors, es denota per $(\vec{F}, \vec{R}, \vec{V'})$ la base de l'espai de càmera, on els vectors són:*

- *El vector direcció de la càmera $\vec{F} = \frac{L-D}{\|L-D\|}$*
- *El vector de la dreta $\vec{R} = \frac{V \times F}{\|V \times F\|}$*
- *El vector vertical rectificat $\vec{V'} = \vec{F} \times \vec{R}$*

Amb aquesta base es pot definir el canvi de base entre coordenades de món i coordenades de càmera:

$$\begin{aligned} C_{view} : \mathbb{R}^3_{mon} &\longrightarrow \mathbb{R}^3_{camera} \\ p_{mon} &\to M_{view} \cdot p_{mon} = p_{camera} \end{aligned} \tag{3.3.5}$$

On:

$$M_{view} = \begin{pmatrix} R_x & R_y & R_z & 0 \\ V'_x & V'_y & V'_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -L_x \\ 0 & 1 & 0 & -L_y \\ 0 & 0 & 1 & -L_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} R_x & R_y & R_z & -L \cdot R \\ V'_x & V'_y & V'_z & -L \cdot V' \\ F_x & F_y & F_z & -L \cdot F \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.3.6}$$

S'observa que la rotació (matriu esquerra) i la translació (matriu dreta) estan invertides (transposada i negada respectivament) ja que es vol rotar i traslladar el món en la direcció oposada a on es vol que la càmera es mogui.

3.3.3 Projecció

En aquesta secció s'explicarà el procés de projectar els punts de l'espai 3D a la pantalla. Per aquest motiu, s'introduiran els conceptes de coordenades homogènies, coordenades de tall i matriu de projecció.

Coordenades homogènies

Definició 3.3.5. *Siguin $p \in \mathbb{R}^3$ un punt i $\vec{v} \in \mathbb{R}^3$ un vector. La posició d'aquest punt en coordenades cartesianes serà*

$$p = (p_1, p_2, p_3) \tag{3.3.7}$$

I el vector serà:

$$\vec{v} = (v_1, v_2, v_3). \tag{3.3.8}$$

Per a poder representar tant vectors com punts en el mateix marc, es pot estendre l'espai 3D a l'espai projectiu 4D si s'introdueixen ***coordenades homogènies****.*

En coordenades homogènies tant els punts 3D (posicions) com els vectors 3D (direccions) són representats com a vectors en 4D, però difereixen en la quarta coordenada. Més concretament, p i $\vec{v}$ són:

$$\begin{aligned} p_h = (t \cdot p_1, t \cdot p_2, t \cdot p_3, t), t \neq 0 \\ \vec{v}_h = (v_1, v_2, v_3, 0) \end{aligned} \tag{3.3.9}$$

En coordenades homogènies t és un factor que permet representar punts a l'infinit (quan $t = 0$), i habitualment es sol considerar que $t = 1$ sempre que sigui un punt que no estigui a l'infinit. Per tant, la representació de p_h seria $p_h = (p_1, p_2, p_3, 1)$.

Amb això, es tenen les eines necessàries per a poder fer transformacions afins i canvis de coordenades. Siguin el sistema A i el sistema B dos sistemes de coordenades. Es representa un punt en el sistema A com p_A i un altre punt en el sistema B p_B, i aquests punts tenen l'expressió $p_A = (x_A, y_A, z_A)$, $p_B = (x_B, y_B, z_B)$. Es defineix un vector de translació $\vec{t} = (t_x, t_y, t_z)$. Si es vol passar del sistema A al B i fer la translació, s'hauria de fer:

$$p_B = \mathbf{M} \cdot p_A + \vec{t} \tag{3.3.10}$$

On $\mathbf{M}$ és una matriu 3×3 que representa les transformacions lineals entre el sistema A i el B. S'observa que $\mathbf{M}$ i $\vec{t}$ es poden combinar en una matriu 4×4 $\mathbf{N}$:

$$\mathbf{N} = \begin{pmatrix} \mathbf{M} & \vec{t} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} & M_{13} & t_x \\ M_{21} & M_{22} & M_{23} & t_y \\ M_{31} & M_{32} & M_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.3.11}$$

En aquesta situació, tant p_A com p_B es transformen en coordenades homogènies afegint la coordenada t, que, com ja s'ha dit abans, és $t = 1$ per a un punt i $t = 0$ per a un vector.

Per tant, quan es multiplica un punt p_A per $\mathbf{N}$ la coordenada t es queda igual i es segueix tenint un punt, i quan es multiplica un vector $\vec{t}$ per $\mathbf{N}$, el resultat és un vector.

Frust, clip coordinates i matriu de projecció

La regió visible de l'escena que es mostra per pantalla és capturada pel frust, una piràmide rectangular truncada que té la càmera a l'àpex, tal i com s'ha definit a la definició 3.3.1, i que constitueix el camp de visió de la càmera. Per passar d'una escena 3D a un pantalla 2D, s'ha de fer una projecció per transformar el frust i els objectes que conté, de tal manera que els objectes més llunyans es vegin més petits. Aquesta transformació es guarda a una matriu anomenada **projectionMatrix** o matriu de projecció.

El resultat d'aquesta transformació són el que s'anomenen **clip coordinates**, il·lustrades a la figura 3.4, que es situen dins d'un espai 4D anomenat **clip space**. Els vèrtexs d'aquest espai tenen coordenades (x', y', z', t'), derivades d'aplicar la matriu de projecció al punt $(x, y, z, 1)$. S'observa que un cub de costat $2t'$ centrat a l'origen encapsula tots els punts que es dibuixaran a pantalla. Aquest cub es defineix com $-t' \leq x', y', z' \leq t'$ dins del *clip space*. Els punts que es queden fora del cub són eliminats abans de rasteritzar.

Finalment, es converteixen les *clip coordinates* en **Normalized Device Coordinates** (NDC) o coordenades normalitzades, també il·lustrades a la figura 3.4, dividint x', y' i z' per t', amb la qual cosa els punts queden així: $(x'', y'', z'', 1)$. Aquesta operació fa que el *clip space* es converteixi en un cub unitat.

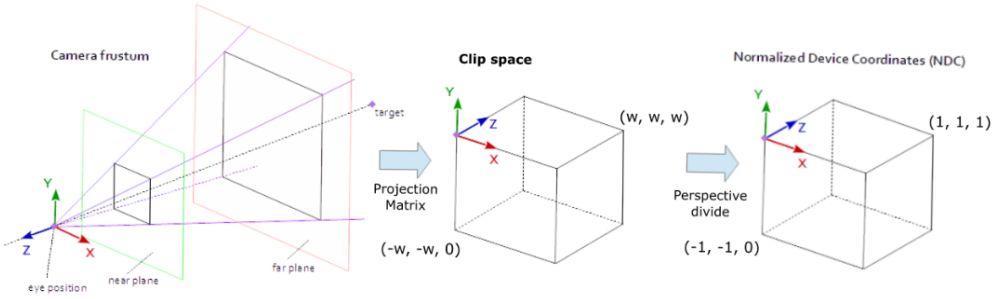


Figura 3.4: Figura on es veu com es passa del frust al NDC, passant pel *clip space*. Imatge extreta de [7]

.

A continuació, s'explica la construcció de la matriu de projecció, basada en el llibre de Eric Lengyel[32]. Quan es projecta un punt $p = (x, y, z)$ al pla de projecció a distància g de la càmera, es té que:

$$x_{proj} = \frac{gx}{z}, y_{proj} = \frac{gy}{z}, z_{proj} = ?, t_{proj} = ?$$

On $y_{proj} \in [-1, 1]$, però $x_{proj} \in [-s, s]$, per tant, s'haurà de dividir la coordenada x per s. No es poden utilitzar directament aquestes equacions, ja que no hi ha transformacions que resultin en una divisió. Tanmateix, es pot utilitzar el fet de que per passar a NDC

s'ha de dividir per la component t. Això dona la següent transformació en coordenades homogènies:

$$\begin{pmatrix} \frac{g}{s} & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{gx}{s} \\ gy \\ Az + B \\ z \end{pmatrix} = p_{clip}$$

Els paràmetres A, B estan per evitar un valor constant a la coordenada z. En el punt de clip space p_{clip}, quan es passa a NDC tenim que el valor de $z_{ndc} \in [0, 1]$, com $\frac{z_{clip}}{t_{clip}} = A + \frac{B}{z}$, llavors quan $z = d$, $A + \frac{B}{d} = 0$ i quan $z = f$, $A + \frac{B}{f} = 1$. Això dona dues equacions, que es poden resoldre per trobar que:

$$A = \frac{f}{f-d}, B = \frac{-df}{f-d} \tag{3.3.12}$$

Per tant, la matriu de projecció és:

$$M_{proj} = \begin{pmatrix} \frac{g}{s} & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \frac{f}{f-d} & \frac{-df}{f-d} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.3.13}$$

I el punt final projectat és:

$$p_{clip} = M_{proj} \cdot p = \begin{pmatrix} \frac{g}{s} & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \frac{f}{f-d} & \frac{-df}{f-d} \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{gx}{s} \\ gy \\ \frac{f}{f-d}z - \frac{df}{f-d} \\ z \end{pmatrix}$$

3.4 Space-Filling Curves (SFC)

Un cop definits els models de dades (*point clouds* i octrees) i la càmera, el problema següent és com tenir els punts ordenats per tal d'optimitzar la visualització de les dades des de una certa posició (o càmera). En aquesta secció es defineixen les estructures d'ordenació espacial que es van introduir a la secció 2.2, concretament, l'ús de la corba de Morton, corba de Lebesgue o corba d'ordre Z. A continuació definirem aquesta corba, les seves propietats i analitzarem com calcular-la i com reordenar-la en funció de la posició de la càmera. Primer, es defineixen alguns conceptes bàsics, com què és una corba i què significa que una corba ompli un espai. Per fer les següents definicions i demostracions ens hem basat en aquest llibre [6] de Michael Bader.

Definició 3.4.1. *Sigui $f : I \longrightarrow \mathbb{R}^n$ una funció contínua que surt d'un compacte $I \subset \mathbb{R}$ cap a $\mathbb{R}^n$. La imatge $f(I)$ de f s'anomena* ***corba****, i $x = f(t), t \in I$ s'anomena* ***representació paramètrica*** *de la corba.*

A continuació es defineix la mesura de Lebesgue, concepte necessari que a poder parlar 'd'omplir' un espai.

Definició 3.4.2. *Sigui $C = I_1 \times ... \times I_n \subset \mathbb{R}^n$ rectangle n-dimensional, definim el seu* ***volum*** *com $vol(C) = l(I_1) \times ... \times l(I_n)$, on $l(I_i) = b_i - a_i$ si $I_i = [a_i, b_i]$ (pot ser obert o tancat).*

Definició 3.4.3. *Sigui $E \subset \mathbb{R}^n$, definim la* ***mesura exterior de Lebesgue*** *com*

$$\lambda^*(E) = inf\{\sum_{k=1}^{\infty} vol(C_k)\} \tag{3.4.1}$$

$$(C_k)_{k\in\mathbb{N}} \text{ és una successió de productes d'intervals oberts amb } E \subset \bigcup_{k=1}^{\infty} C_k \tag{3.4.2}$$

Si a més a més E compleix que $\lambda^*(A) = \lambda^*(A \cap E) + \lambda^*(A \cap E^c) \forall A \subset \mathbb{R}^n$, direm que E és ***Lebesgue mesurable*** *i la seva* ***mesura de Lebesgue*** *serà $\lambda(E) = \lambda^*(E)$.*

Amb aquests conceptes definits, ara es pot passar a definir què és una SFC:

Definició 3.4.4. *Sigui $f(I)$ una corba i sigui $f : I \longrightarrow \mathbb{R}^n$ la seva funció associada, llavors $f(I)$ és una* ***corba que omple l'espai*** *(SFC, de l'anglès Space-Filling Curve) si $f(I)$ té mesura de Lebesgue major que 0.*

La generació de SFCs impliquen recursivitat, per la qual cosa s'introdueix el concepte de corba recursiva:

Definició 3.4.5. *Una SFC $f : I \longrightarrow Q \subset \mathbb{R}^n$ s'anomena* ***recursiva*** *si existeix una partició en diferents nivells de I i de Q en subintervals $I_{l,m}$ i subdominis (oberts connexos) $Q_{l,m}$ amb $I_{0,0} = I$ i $Q_{0,0} = Q$ tals que:*

- *A cada nivell $l > 0$, els intervals $I_{l,m}$ i els subdominis $Q_{l,m}$ són resultat d'una partició dels intervals I_{l-1,m^*} i els subdominis $Q_{l-1,m*}$ en $M(l)$ parts iguals.*
- *Els subdominis $Q_{l,m}$ són congruents a $Q_{l-1,m*}$ si s'escalen amb el factor adequat.*
- *Els subintervals $I_{l,m}$ s'envien als seus subdominis corresponents $Q_{l,m}$, és a dir, que $f(I_{l,m}) = Q_{l,m} \forall l, m$*

Definició 3.4.6. *Sigui $f : I \longrightarrow Q \subset \mathbb{R}^n$ una SFC recursiva. Anomenem* ***ordre*** *de la corba al nombre de vegades que s'ha iterat la recursivitat.*

3.4.1 Corbes contínues

Existeixen moltes corbes que omplen l'espai i són contínues, com ara la corba de Hilbert o la corba de Peano. Per a la visualització de núvols de punts, com s'ha comentat a la secció 2.2, la corba de Hilbert resulta interessant per les seves propietats de localitat. En aquesta secció es defineix la corba de Hilbert i es demostrarà que és una corba contínua.

Per a definir la corba de Hilbert, il·lustrada a la figura 3.5, s'utilitzaran successions d'intervals 1D i 2D. La idea fonamental darrera d'aquesta definició és que la corba de

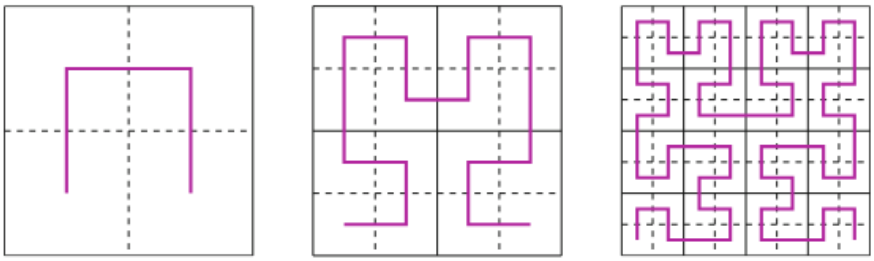


Figura 3.5: Les primeres tres iteracions de la corba de Hilbert. Imatge extreta de [6]

.

Hilbert ha de recórrer el quadrat unitat a velocitat constant, és a dir, quan ha cobert una quarta part del quadrat es vol que s'hagi recorregut una quarta part de l'interval de sortida, i que aquesta propietat es mantingui per la recursió.

Per tant, sigui $t \in [0,1]$, es construeix una successió d'intervals de la forma següent:

- Per l'interval inicial, s'agafa $[0,1]$.
- Per a cada interval $[a, a+h]$, el seu successor és un dels intervals

$$[a, a+\frac{h}{4}], [a+\frac{h}{4}, a+\frac{2h}{4}], [a+\frac{2h}{4}, a+\frac{3h}{4}], [a+\frac{3h}{4}, a+h] \tag{3.4.3}$$

  Concretament, el que contingui el paràmetre t.

Per exemple, per a $t = \frac{1}{3}$, s'obtindrà la successió: $[0,1], [\frac{1}{4}, \frac{2}{4}], [\frac{5}{16}, \frac{6}{16}], [\frac{21}{64}, \frac{22}{64}]$...

Seguint la idea inicial, cada interval d'aquesta seqüència té com a imatge un quadrat específic. De forma anàloga als intervals anteriors, es construeix una successió de quadrats, on cada subquadrat serà un quart del quadrat original $[a,b] \times [c,d]$:

$$[a, \frac{a+b}{2}] \times [c, \frac{c+d}{2}], [\frac{a+b}{2}, b] \times [c, \frac{c+d}{2}], [a, \frac{a+b}{2}] \times [\frac{c+d}{2}, d], [\frac{a+b}{2}, b] \times [\frac{c+d}{2}, d] \tag{3.4.4}$$

Aquesta assignació està il·lustrada a la figura 3.6.

Definició 3.4.7. *Es defineix la* ***corba de Hilbert*** *com la imatge de la funció* $h(t) : [0,1] \to [0,1]^2$ *tal que:*

- *Per a cada paràmetre* $t \in [0,1]$ *existeix una successió d'intervals* $[0,1] \supset [a_1, b_1] \supset ... \supset [a_n, b_n] \supset ...$ *tal que cada interval s'obté dividint l'interval anterior en 4 parts iguals com a l'equació 3.4.3.*
- *Cada successió d'intervals es pot mapejar una a una amb una successió de quadrats 2D com els de 3.4.4. Les iteracions de la corba de Hilbert determinen quin subquadrat és la imatge de quin interval.*
- *La successió d'intervals 2D convergeix a un únic punt a* $Q = [0,1]^2$*, que serà* $h(t)$*.*

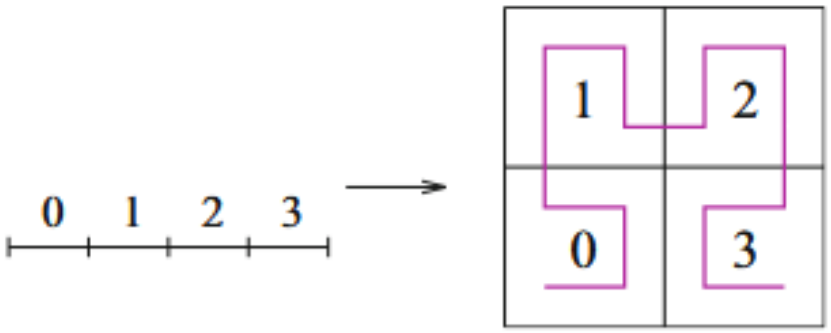


Figura 3.6: Representació de com s'assignen subintervals a subquadrats, utilitzant la primera iteració de la corba de Hilbert. Imatge extreta de [6]

.

A continuació es demostrarà que aquesta corba no només és contínua, sinó que és contínua uniforme:

Donats $t_1, t_2 \in [0,1]$, es pot escollir $n \in \mathbb{N}$ tal que $|t_1 - t_2| < 4^{-n}$. Per a aquesta n, la longitud màxima dels intervals és 4^{-n}. Per tant, l'interval $[t_1, t_2]$ pot estar en com a màxim 2 intervals diferents i contigus. Degut a la construcció de la corba que s'ha fet, dos intervals qualssevol que siguin contigus estan mapejats a dos quadrats que comparteixen un costat. Per construcció, aquests quadrats tenen costat de 2^{-n} de longitud. Les imatges $h(t_1)$ i $h(t_2)$ estaran per tant en un rectangle format pels dos quadrats. La distància entre $h(t_1)$ i $h(t_2)$ ha de ser més petita que la diagonal d'aquest rectangle. Els costats del rectangle mesuren 2^{-n} i $2 \cdot 2^{-n}$, per tant la diagonal és de $\sqrt{5} \cdot 2^{-n}$ (teorema de Pitàgores). Aquesta longitud és una cota superior per la distància entre $h(t_1)$ i $h(t_2)$: $\|h(t_1) - h(t_2)\|_2 < \sqrt{5} \cdot 2^{-n}$.

Per a qualsevol $\epsilon > 0$ sempre es pot escollir $n \in \mathbb{N}$ tal que $\epsilon > \sqrt{5} \cdot 2^{-n}$. Amb aquesta n s'obté una $\delta = 4^{-n}$ tal que per a tot $t_1, t_2 \in [0,1]$ amb $|t_1 - t_2| < \delta$ es té $\|h(t_1) - h(t_2)\|_2 < \sqrt{5} \cdot 2^{-n} < \epsilon$. Per tant, h és contínua uniforme.

3.4.2 Corbes discretes

Notació 2. *En aquest treball denotarem com a* $0_i.b_1b_2b_3...$ *al nombre representat per l'expressió decimal* $0.b_1b_2b_3...$ *en base* i.

L'ordre de Morton (Z-order), il·lustrat a la figura 3.7, anomenat també corba de Lebesgue en el context discret, defineix una ordenació espacial mitjançant la intercalació dels bits de les coordenades.

Considerem la següent funció que surt de l'interval $[0,1]$ cap al quadrat unitat $[0,1] \times [0,1] \subset \mathbb{R}^2$:

$$m(t) = m(0_2.b_1b_2b_3b_4b_5b_6...) = \begin{pmatrix} 0_2.b_1b_3b_5... \\ 0_2.b_2b_4b_6... \end{pmatrix} \tag{3.4.5}$$

Aquesta funció s'anomena *ordre de Morton*, especialment en informàtica. Tots els

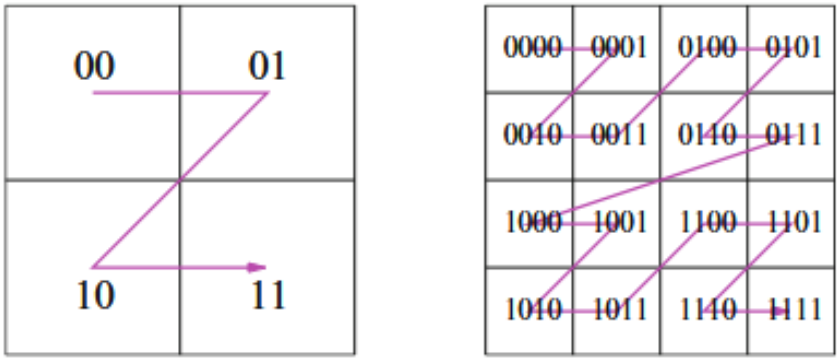


Figura 3.7: Primeres dues iteracions de la corba de Lebesgue. Imatge extreta de [6]

.

punts que comencen amb la mateixa seqüència de n dígits estan al mateix subquadrat de costat 2^{-n}. És evident veure que la funció associada $m(t)$ a l'ordre de Morton no és una funció contínua, per exemple, té una discontinuïtat quan $t = \frac{1}{2}$. En aquest punt, el límit per l'esquerra és:

$$\lim_{n\to\infty} m(0_2.0\underbrace{111\ldots111}_{2n-1 \text{ times}}) = \lim_{n\to\infty} \begin{pmatrix} 0_2.011...1 \\ 0_2.\underbrace{111\ldots1}_{n \text{ times}} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} \tag{3.4.6}$$

Mentre que el límit per la dreta és:

$$m(0_2.1000...) = \lim_{n\to\infty} \begin{pmatrix} 0_2.1000... \\ 0_2.000\ldots \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix} \tag{3.4.7}$$

Com $m(t)$ no és contínua, no pot ser una corba, ni una corba que ompli l'espai. Tanmateix, si es canviés l'espai de sortida per un espai on, a cada salt de $m(t)$, hi hagués un salt, la funció llavors sí que seria contínua. A continuació s'introdueix el *conjunt de Cantor*, un dels possibles conjunts que compleixen aquesta propietat:

Definició 3.4.8. *El **conjunt de Cantor** és el conjunt de tots els nombres $t \in [0,1]$ que poden ser escrits en base 3 amb la següent presentació:*

$$t = 0_3.t_1t_2t_3..., t_i \in \{0,2\} \tag{3.4.8}$$

Aquest conjunt es representa amb $\mathcal{C}$.

Ara ja s'han introduït totes les eines necessàries per a poder definir la corba de Lebesgue o de Morton:

Definició 3.4.9. *Anomenem **corba de Lebesgue o de Morton** a la imatge de la següent funció:*

$$Z : \mathcal{C} \to [0,1]^2, Z(t) = Z(0_3.t_1t_2t_3t_4...) = \begin{pmatrix} 0_2.\frac{t_1}{2}\frac{t_3}{2}... \\ 0_2.\frac{t_2}{2}\frac{t_4}{2}... \end{pmatrix} \tag{3.4.9}$$

Aquesta corba és exhaustiva, ja que per a cada punt del quadrat unitat

$$p(t) = \begin{pmatrix} 0_2.b_1b_2b_3... \\ 0_2.d_1d_2d_3... \end{pmatrix} \in [0,1]^2 \tag{3.4.10}$$

és la imatge del paràmetre $t = 0_3.(2b_1)(2d_1)(2b_2)(2d_2)(2b_3)(2d_3)...$, que pertany al conjunt de Cantor.

Notem que aquesta definició és per la corba de Morton en el pla, però podem generalitzar aquesta definició a qualsevol dimensió $n > 1$:

Definició 3.4.10. *Anomenem* ***corba de Lebesgue o de Morton*** *n-dimensional a la imatge de la següent funció:*

$$Z^n : \mathcal{C} \to [0,1]^n, Z^n(t) = Z^n(0_3.t_1t_2t_3t_4...) = \begin{pmatrix} 0_2.\frac{t_1}{2}\frac{t_{n+1}}{2}... \\ 0_2.\frac{t_2}{2}\frac{t_{n+2}}{2}... \\ ... \\ 0_2.\frac{t_n}{2}\frac{t_{2n}}{2}... \end{pmatrix} \tag{3.4.11}$$

A continuació demostrarem que aquesta corba és contínua per a tot $t_o \in \mathcal{C}$ en el cas $n = 2$:

Sigui $t \in \mathcal{C}$ un paràmetre amb $|t - t_0| < 3^{-2n}$, on 3^{-2n} és l'amplada del terç mitjà que s'esborra a la iteració $2n$ de la construcció del conjunt de Cantor. Si t_0 i t estan a menys de 3^{-2n} de distància no pot haver cap buit entre ells. Per tant, els primers $2n$ dígits ternaris han de ser els mateixos:

$$\begin{aligned} t_0 &= 0_3.t_1t_2...t_{2n}... \\ t &= 0_3.t_1t_2...t_{2n}... \end{aligned} \tag{3.4.12}$$

Les seves imatges corresponents són:

$$\begin{aligned} Z(t_0) &= \begin{pmatrix} 0_2.t_1t_3...t_{2n-1} \\ 0_2.t_2t_4...t_{2n} \end{pmatrix} \\ Z(t) &= \begin{pmatrix} 0_2.t_1t_3...t_{2n-1} \\ 0_2.t_2t_4...t_{2n} \end{pmatrix} \end{aligned} \tag{3.4.13}$$

Els dos punts comparteixen els primers n dígits i per tant estan en quadrat de costat 2^{-n}. La seva distància màxima és per tant $\|Z(t) - Z(t_0)\| < 2^{-n}\sqrt{2}$. Per a cada $\varepsilon > 0$ podem escollir n tal que $2^{-n}\sqrt{2} < \varepsilon$. Amb $\delta = 3^{-2n}$ podem inferir que per a tota t tal que $|t - t_0| < \delta$ la desigualtat $\|Z(t) - Z(t_0)\| < 2^{-n}\sqrt{2}$ es compleix. Això implica la continuïtat de Z a t_0. Com t_0 es escollida arbitràriament Z és contínua a tota $\mathcal{C}$.

Algorisme de construcció del codi Morton

En aquest apartat es descriurà l'algorisme utilitzat per a ordenar un *point cloud* seguint la corba de Lebesgue i es calcularà el seu cost. La implementació d'aquest algorisme es detallarà a la secció 4.2. Com es pot veure, l'algorisme 1 consta bàsicament de 4 etapes, que són: 1) Transformar els punts del *point cloud* per a que siguin enters positius, 2) Calcular el codi Morton per a cada punt, 3) Ordenar els punts d'acord a aquest ordre, 4) Crear el nou *point cloud* amb els punts ordenats. A continuació s'explicarà cada etapa calculant quin és el seu cost:

- **1. Transformació dels punts**: Aquesta etapa correspon a la funció en pseudocodi *transformToInt*, i el que fa és escalar els punts per a que siguin enters positius, utilitzant operacions aritmètiques (sumes, restes, multiplicacions i divisions), que tenen cost $O(d)$ per la suma/resta i $O(d^2)$ per multiplicacions, on d és el nombre de dígits. Aquestes operacions es realitzen a cada coordenada de cada punt, per tant, en el cas tridimensional s'estaran fent les operacions $3n$ vegades, el que dona $O(3d^2n) = O(n)$, ja que 3 és constant i d és en teoria molt més petit que n.
- **2. Càlcul del codi Morton**: Correspon a la funció de pseudocodi *calcMortonCode*. La seva implementació s'explicarà amb més detall a la secció 4.2, però bàsicament utilitza manipulació de bits per intercalar els bits de les tres coordenades de cada punt. Aquestes operacions amb bits escalen amb el nombre de bits, però com es treballa amb *int* de 32 bits, aquest cost serà constant: $O(1)$. S'apliquen aquestes operacions a cada coordenada de cada punt, per tant, la complexitat d'aquesta etapa és $O(3n) = O(n)$.
- **3. Ordenació dels punts**: Correspon a les funcions de pseudocodi *indexThatSortList* i *order*. La primera funció s'encarrega d'ordenar la llista *codis* i retornar els índexs que la ordenen, i la segona ordena la llista *punts* segons els índexs. Per a la primera funció, els algorismes d'ordenació més ràpids, com *quicksort*, *heapsort* o *mergesort* tenen complexitat $O(n \cdot log(n))$. Per a la segona, és només anar iterant per la llista *índexos* ($O(n)$) mentre s'accedeix a la llista *punts* ($O(1)$) i es copien els elements a *puntsSorted* ($O(1)$). El cost d'aquesta etapa serà per tant $O(n \cdot log(n))$.
- **4. Creació del point cloud**: Correspon a la funció de pseudocodi *createPointCloud*. Per crear el *point cloud* només s'ha d'iterar per la llista de punts, que té complexitat $O(n)$.

La complexitat total de l'algorisme és la més alta de les seves etapes, per tant, $O(n \cdot log(n))$.

3.4.3 Preservació de localitat

Com s'ha dit anteriorment, les SFC preserven la localitat dels punts. En aquesta secció s'explica què vol dir preservar la localitat, i s'introdueixen dues mesures diferents per calcular la localitat.

Definició 3.4.11. *Es diu que una corba* ***preserva la localitat*** *dels punts si, per a dos punts qualssevol de l'interval que estan a poca distància les seves imatges també estan a*

Algorithm 1 Codi de Morton

```
Input: Point cloud:pc
Output: Point cloud ordenada:pcMorton
function MAIN(pc)
    punts = pc.getPoints()
    puntsInt = transformToInt(punts)
    codis = calcMortonCode(puntsInt)
    indexos = indexThatSortList(codis)
    puntsSorted = order(punts, indexos)
    pcMorton = createPointCloud(puntsSorted)
    return pcMorton
end function
```

poca distància, i viceversa. Aquesta és la propietat més rellevant pel paral·lelisme, l'accés a memòria i coherència espacial en visualitzar.

A la literatura, s'han definit diversos mètodes per quantificar aquesta propietat, i la corba de Lebesgue acostuma a tenir bon rendiment. Per aquesta secció, s'utilitzen dues definicions de localitat. La primera prové d'aquest estudi [10], que utilitza la següent mesura de localitat:

$$L_\delta(C) = \sum_{i,j \in [n^m] | i<j \text{ and } d_1(C(i),C(j))=\delta} |i-j|. \tag{3.4.14}$$

On C és una corba, $[n^m]$ és una graella m-dimensional de mida $n \times n \times ... \times n$ i d_1 és la distància de Manhattan o norma l_1 ($d_1(\mathbf{p},\mathbf{q}) = \sum_{i=1}^{n} |p_i - q_i|$).

Aquesta funció $L_\delta(C)$ acumula totes les diferències d'índexs de parelles de punts que es troben una distància δ, utilitzant la distància d_1. Aquest estudi acaba concloent el següent per a $\delta = 1$:

- Per a la corba de Lebesgue en 2D amb ordre k (és a dir, que s'ha aplicat la recursió k vegades): $L_1(Z_k^2) = 2^{3k} - 2^k$.
- Per a la corba de Lebesgue en 3D amb ordre k: $L_1(Z_k^3) = 2^{5k} - 2^{2k}$.

També afirmen que, per a $\delta << 2^k$ i $m = 2, 3$, la mesura de localitat L_δ de la corba de Lebesgue és millor que la corba de Hilbert.

La segona definició de localitat utilitzada està basada en aquesta web [14]. En aquest cas la mesura de localitat utilitzada és la mitjana de totes les distàncies euclidianes ($d_2(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$) entre punts consecutius. Per tenir uns valors experimentals d'aquesta mesura, s'han generat corbes de Morton (en 2D i en 3D) de diferents ordres i després s'han fet les distàncies entre punts consecutius i calculat la mitjana. S'ha provat amb diferents nombres de punts (sempre potències de 4 pel cas 2D i potències de 8 pel cas 3D, per l'estructura de quadtree i octree implícita respectivament). Els resultats d'aquests càlculs es poden trobar a les taules 3.1 i 3.2.

Primer de tot, s'observa que la mesura de localitat és més gran a la corba tridimensional que a la corba plana. Això té molt sentit, ja que a la corba plana la distància entre punts és l'arrel quadrada de dos sumands, mentre que a la corba 3D són tres sumands, això fa que la distància mitjana sigui més alta. També s'observa que, a mesura que es va iterant la corba, en els dos casos la distància mitjana augmenta. Això es deu al fet que, com més s'itera la corba, van apareixent salts molt grans (i poc freqüents), que esbiaixen la mitjana cap a dalt. En el cas 2D observem que el creixement de la distància mitjana sembla frenar-se a mesura que l'ordre augmenta, i veiem que entre ordre 10 i 11 la mesura creix només un $0,06\%$, per tant, es veu una convergència quan l'ordre de la corba tendeix a infinit. En el cas 3D es veu que entre ordre 7 i 8 la diferència és d'un $0,065\%$, per tant, també es veu una convergència quan l'ordre de la corba tendeix a infinit.

Ordre de la corba	Nombre de punts	Mesura localitat
1	4	1.1381
2	16	1.3098
4	256	1.5573
7	16384	1.6570
10	1048576	1.6705
11	4194304	1.6715

Taula 3.1: Taula que mostra el nombre d'iteracions o ordre de la corba, la quantitat de punts i la mesura de localitat per a la corba de Lebesgue en 2D

Ordre de la corba	Nombre de punts	Mesura localitat
1	8	1.3254
2	64	1.7586
4	4096	2.0980
7	2097152	2.1497
8	16777216	2.1511

Taula 3.2: Taula que mostra el nombre d'iteracions o ordre de la corba, la quantitat de punts i la mesura de localitat per a la corba de Lebesgue en 3D

3.4.4 Relació amb quadtrees i octrees

Tant l'ordre de Morton (o ordre Z) com el de Hilbert permeten induir estructures jeràrquiques de subdivisió de l'espai, com quadtrees en 2D i octrees en 3D [49], mitjançant la discretització prèvia de les coordenades. En el cas de l'ordre Z, la corba de Morton en $\mathbb{R}^n$ es pot utilitzar per crear un arbre on cada node té 2^n fills. La idea principal és ordenar el conjunt de punts d'entrada d'acord amb l'ordre de la corba i, un cop ordenat, es pot guardar aquest conjunt dins d'una estructura com un arbre. Recórrer els punts segons l'ordre Z és equivalent a recórrer un arbre N-dimensional amb un algorisme de profunditat (DFS, o Depth-First Search) en preordre. Els punts del point cloud d'entrada s'han d'escalar per a que totes les seves coordenades siguin enters positius. Aquest algorisme es pot veure a l'algorisme 2.

Algorithm 2 Construcció d'un octree a partir de l'ordre Z (Morton)

Input: Conjunt de punts $P = \{p_i = (x_i, y_i, z_i)\}$, profunditat màxima k
Output: Octree **O** que conté tots els punts de P
function BUILD_OCTREE(P)
    Escalar i quantitzar les coordenades de cada punt a enters de k bits
    for $p_i \in P$ **do**
        $\tilde{x}_i \leftarrow \lfloor x_i \cdot 2^k \rfloor$
        $\tilde{y}_i \leftarrow \lfloor y_i \cdot 2^k \rfloor$
        $\tilde{z}_i \leftarrow \lfloor z_i \cdot 2^k \rfloor$
        $M(p_i) \leftarrow \text{interleaveBits}(\tilde{x}_i, \tilde{y}_i, \tilde{z}_i)$
    end for
    Ordenar els punts P segons el codi Morton $M(p_i)$ (usant l'algorisme 1)
    Inicialitzar l'arrel r de l'octree **O**
    for $p_i \in P$ en ordre creixent de $M(p_i)$ **do**
        $n \leftarrow r$
        for $l = 0$ to $k - 1$ **do**
            Extreure els bits (b_x^l, b_y^l, b_z^l) del nivell l
            $c \leftarrow b_x^l + 2b_y^l + 4b_z^l$
            if el fill $n.c$ no existeix **then**
                Crear el node fill $n.c$
            end if
            $n \leftarrow n.c$
        end for
        Inserir el punt p_i al node n
    end for
    return O
end function

3.4.5 Reordenació respecte a la posició de la càmera

Donat que l'ordre de Morton indueix implícitament una estructura d'octree, és possible recórrer el conjunt de punts segons diferents ordres de visita dels suboctants. En particular, l'ordre dels fills de cada node es pot adaptar en funció de la posició de la càmera, permetent un recorregut que prioritzi els suboctants més propers. Aquest enfocament és equivalent a aplicar l'algorisme del pintor sobre l'arbre implícit, tot i que no garanteix una ordenació correcta de la profunditat en tots els casos (per exemple quan hi ha perspectiva o quan la mida de la projecció de la primitiva associada al punt és més gran que un píxel).

Seguint l'algorisme 1 del codi Morton, per construir el codi Morton en 3D, es passen totes les coordenades dels punts a enters positius, i es van intercalant els bits, de tal manera que un punt amb coordenades en binari $p = (x_1x_2..., y_1y_2..., z_1z_2...)$ tindrà s'assignat el codi $x_1y_1z_1x_2y_2z_2...$ en binari. S'observa que els grups de 3 bits divideixen l'espai en 8 cubs, tal i com es pot veure a la figura 3.8. Si hi hagués una càmera mirant de cara al cub 111 i es recorregués el codi en ordre ascendent, es començaria al cub 000, 001, 010, ..., 111. Es nota que, en aquest ordre, els cubs només poden "tapar"als cubs que van abans i només poden ser "tapats"pels cubs que venen després. Aquest ordre es troba il·lustrat a la figura 3.9.

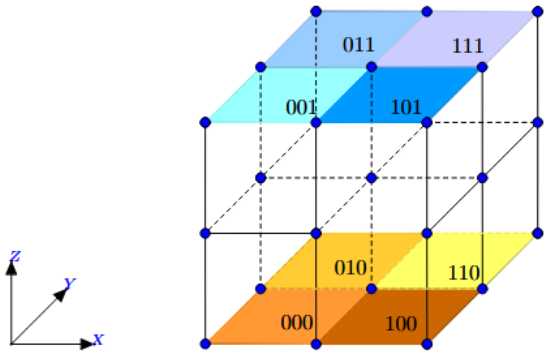


Figura 3.8: Divisió de l'espai en vuit regions cúbiques en funció dels 3 bits més significatius del codi Morton. Es pot veure l'estructura d'octree implícita. Imatge extreta de [19]

.

Cal destacar que aquest recorregut no resol directament la visibilitat, ja que aquesta és una propietat local que es decideix a nivell de píxel, però sí que agrupa punts amb profunditats similars i redueix la probabilitat que punts molt allunyats competeixin entre si durant el càlcul de la profunditat. En conseqüència, l'ordenació basada en l'octree implícit no substitueix el Z-buffer, però el complementa de manera efectiva: el recorregut jeràrquic redueix el nombre de conflictes de visibilitat i el Z-buffer resol de forma exacta els casos residuals. Aquesta combinació permet millorar l'eficiència global del renderitzat sense comprometre la correcció del resultat final.

En casos on la perspectiva està desactivada, les primitives associades als punts es projecten en un únic punt i la càmera està alineada amb els eixos coordenats, aquesta ordenació permetria desactivar del tot el Z-buffer i aplicar l'algorisme del pintor.

A continuació, a la taula 3.3 s'explica en quin ordre s'han de recórrer els cubs en cas

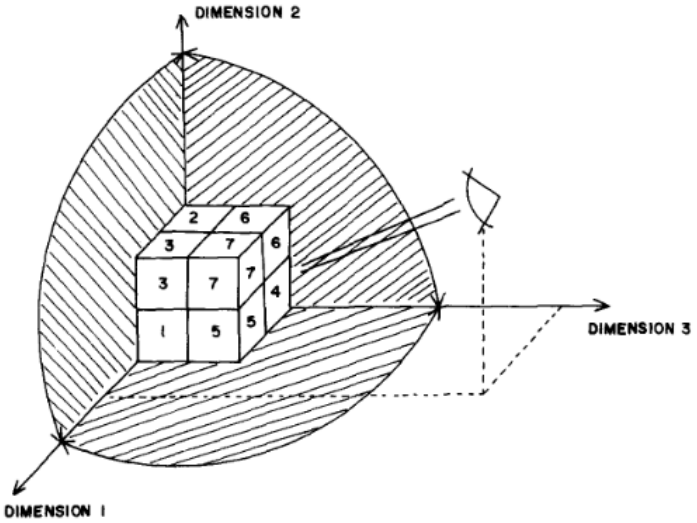


Figura 3.9: Per a una càmera situada en aquest octant, es veu que la seqüència 0, 1, 2, 3, 4, 5, 6, 7 produeix una imatge correcta. Imatge extreta de [37]

.

que es tingués la càmera a cada octant. Hi ha diversos ordres possibles, però només s'explica un per cada cas.

Octant on està la càmera	Ordre per recórrer el codi Morton
Cub 111	Ordre ascendent del codi de Morton
Cub 110	001, 000, 011, 010, 101, 100, 111, 110
Cub 101	010, 011, 110, 111, 000, 001, 100, 101
Cub 100	011, 010, 111, 110, 001, 000, 101, 100
Cub 011	Ordre invers del cub 100
Cub 010	Ordre invers del cub 101
Cub 001	Ordre invers del cub 110
Cub 000	Ordre descendent del codi de Morton

Taula 3.3: Taula que mostra l'ordre en què s'hauria de recórrer la corba de Lebesgue en cas que la càmera estigués a cada octant.

Aquests ordres es poden aplicar de forma recursiva en cas de que la corba tingui ordre $k > 1$ agafant els bits del codi de Morton de 3 en 3 començant pels bits més significatius.

En cas que es tingués un octree definit explícitament, es podria canviar l'algorisme per recórrer l'arbre d'acord a la taula anterior. Tanmateix, l'objectiu d'ordenar els punts amb el codi Morton és precisament no utilitzar aquestes estructures, per tant, s'introdueix a continuació un algorisme que permet reordenar els punts del *point cloud* per poder tenir tots els diferents ordres pels octants.

Algorisme de reordenació de punts

En aquest apartat s'explica l'algorisme que serveix per a reordenar els punts del point cloud depenent de la posició de la càmera (veure algorisme 3) i es calcula el seu cost.

Algorithm 3 Reordenació del codi de Morton

```
Input: Point cloud:pc, int: octant
Output: Point cloud reordenada:pc2
function MAIN(pc, octant)
    punts = pc.getPoints()
    puntsInt = transformToInt(punts)
    codis = calcMortonCode(puntsInt)
    codisReordre = reorderMortonCode(codis, octant)
    tupla = new Array( (codisReordre, punts) )
    tuplaSorted = radixSort(tupla)
    puntsSorted = getPunts(tuplaSorted)
    pc2 = createPointCloud(puntsSorted)
    return pc2
end function
```

Com es pot observar, l'algorisme està comprès per 4 etapes, i són: 1) Crear els codis de Morton per a tots els punts del *point cloud*, 2) Reordenar els codis de Morton depenent de la posició de la càmera, 3) Assignar els codis de Morton als punts i ordenar-los, 4) Crear un *point cloud* amb els punts ordenats. Una alternativa a aquest algorisme seria incloure una llista amb els codis de Morton com a atribut dins del *point cloud*, juntament amb un int per saber l'octant (i per tant l'ordre) que va amb aquells codis, per no haver de calcular-los cada cop, amb això es podria evitar la primera etapa. A continuació es calcula la complexitat d'aquest algorisme:

- **1. Càlcul dels codis de Morton**: Igual que a l'algorisme 1, la complexitat d'aquesta etapa és $O(n)$.
- **2. Reordenació dels punts depenent de la càmera**: Aquesta es correspon amb la funció de pseudocodi *reorderMortonCode*. El que fa aquesta funció és, per a cada punt en codi de Morton, agafa els bits del codi de 3 en 3 començant pels menys significatius, i els canvia segons l'octant on està la càmera, seguint els ordres que es van veure a la secció 3.4.5 (per exemple, si s'estava mirant des del cub 5 i els 3 bits són 010 i ara s'està mirant des del cub 6, els 3 bits canviarien a 001). Com en el projecte s'utilitzen codis de Morton de 30 bits (10 per coordenada), s'està fent l'operació de substitució 10 vegades per punt del *point cloud*, i la complexitat d'aquesta etapa és $O(10n) = O(n)$.
- **3. Assignació dels codis i Radix Sort**: En aquesta etapa s'utilitza una tècnica semblant a una que es veurà a la secció 3.5.2, que és assignar un nombre a cada element i ordenar amb *radix sort*. La diferència és que aquí els nombres assignats no són aleatoris, sinó que es posen expressament per a que sorti l'ordre que es vol. Crear una llista i assignar els elements són $O(n)$ operacions, i el *radix sort* té complexitat $O(n \cdot d)$, on d és el nombre de dígits del nombre, i com s'està treballant amb *int* de 32 bits, d és constant. Per tant, aquesta etapa també és $O(n)$.

- **4. Creació del point cloud**: Igual que a l'algorisme 1, correspon a la funció de pseudocodi *createPointCloud*, i per crear el *point cloud* només s'ha d'iterar per la llista de punts, que té complexitat $O(n)$.

Per tant, aquest algorisme tindria una complexitat $O(n)$.

3.5 Operacions de Permutació i Shuffle

En termes d'eficiència, en paral·lelitzar, sorgeixen problemes de patrons regulars que provoquen que es projectin molts punts de la corba en la mateixa regió de l'espai, fet que provoca divergència i desequilibris de càrrega entre processadors, tal i com s'ha introduït al capítol 2. Per solucionar aquesta regularitat, en aquesta secció s'estudiarà què són les permutacions, s'analitzaran alguns dels algorismes utilitzats per permutar conjunts i s'explicarà com desordenar la corba de Morton pot arribar a millorar el rendiment.

3.5.1 Permutacions

El concepte de permutació sorgeix de la necessitat de definir de forma rigorosa les diferents maneres en que es poden ordenar els elements d'un conjunt. Les permutacions, denotades usualment amb lletres gregues, es poden definir com:

Definició 3.5.1. *Sigui $S = \{s_1, s_2, ..., s_n\}$ un conjunt. Es defineix una **permutació** σ de S com una funció bijectiva de S a si mateix:*

$$\sigma : S \longrightarrow S \tag{3.5.1}$$

Les propietats de les permutacions no depenen del tipus d'element del conjunt, només depenen del seu índex, per tant, es pot considerar que tots els conjunts sobre els que s'apliquen permutacions són $S = \{1, 2, ...n\}$. A continuació s'explicarà la notació que s'utilitzarà per representar permutacions.

Notació 3. *La notació de dues línies llista els elements del conjunt a la primera línia, mentre que a la segona es mostra la seva imatge. Per exemple, una permutació de $S = \{1, 2, 3, 4\}$ tal que*

$$\sigma(1) = 2, \sigma(2) = 4, \sigma(3) = 3, \sigma(4) = 1$$

es representaria així:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$$

Notació 4. *La notació d'una línia s'utilitza quan els elements de S tenen un ordre. En aquesta notació s'omet la primera línia i roman només amb la segona. Per exemple, la permutació anterior es representaria com:*

$$\sigma = 2431$$

Si els elements tinguessin més d'un dígit s'utilitza una coma per separar-los.

3.5.2 Algorismes de shuffle

En aquesta secció s'explicaran dos dels algorismes utilitzats per a fer operacions de *shuffle*: l'algorisme de Fisher-Yates i un algorisme que utilitza el radix sort, i es compararan amb un algorisme *naïve* o ingenu per veure quines avantatges tenen.

Algorisme ingenu

Quan es pensa en com generar una permutació d'un conjunt d'elements, una de les primeres idees en què es podria pensar és posar els elements del conjunt en una llista i per a cada element de la llista, generar un nombre aleatori i intercanviar aquests dos elements en el ordre, i iterar sobre la llista sencera d'aquesta manera. Aquest algorisme, detallat en pseudocodi a l'algorisme 4, té molt bon comportament asimptòtic ($O(n)$), però no és un bon algorisme per fer permutacions, ja que està esbiaixat, és a dir, algunes permutacions tindran més probabilitats d'aparèixer que altres. Això es deu a que el nombre total de permutacions $n!$ no divideix al nombre total de resultats de l'algorisme n^n, per tant, algunes permutacions tindran més resultats assignats. Al següent algorisme la línia *exchange(list.at(i), list.at(j))* és equivalent a aplicar la permutació $\sigma = 1, ..., i-1, j, i+1, ..., j-1, i, j+1, ..., n$.

Algorithm 4 Algorisme ingenu

```
Input: Array de int: list
Output: Array de int reordenada: σ(list)
function NAIVE_SHUFFLE(list)
    n = list.size()
    for i = 0 to n do
        j = getRandomIntegerRange(n)
        exchange(list.at(i), list.at(j))
    end for
end function
```

Fisher-Yates shuffle

L'algorisme de Fisher-Yates 5, il·lustrat a la figura 3.10, és un algorisme de permutar un conjunt finit, molt utilitzat a l'àmbit de la informàtica. L'algorisme té com a entrada una llista amb tots els elements del conjunt, i contínuament determina quin és el següent element de la nova llista agafant un element de la llista original fins que no queda cap element. Aquest algorisme produeix permutacions sense biaix, és a dir, cada permutació té les mateixes probabilitats de ser generada. A més, a l'igual que l'algorisme ingenu, aquest algorisme també té $O(n)$, per tant és molt eficient.

Aquest algorisme pot semblar que és igual a l'algorisme ingenu, però aquest algorisme, mentre va passant per la llista, mai toca les posicions anteriors, amb això fa que el nombre de resultats d'aquest algorisme sigui de $n!$, que és el mateix que el nombre de permutacions de la llista, i per això aquest algorisme no és esbiaixat.

Es vol mencionar que en el projecte, per fer totes les operacions de *shuffle*, s'utilitza

la funció *shuffle* del paquet *random.py* de *Python*, que implementa aquest algorisme [3].

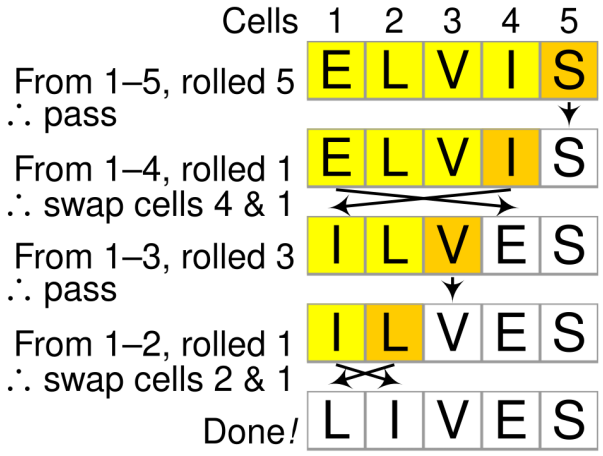


Figura 3.10: Imatge que mostra el funcionament de l'algorisme de Fisher-Yates. Imatge extreta de [8]

.

Algorithm 5 Algorisme de Fisher-Yates

```
Input: Array de int:list
Output: Array de int reordenada:list
function FISHER_YATES_SHUFFLE(list)
    n = list.size()
    for i = n − 1 to 1 do
        j = getRandomIntegerRange(i) //σ(i) = j
        exchange(list.at(i), list.at(j))
    end for
end function
```

Radix sort

Aquest mètode es basa en l'algorisme d'ordenació *radix sort*. En aquest algorisme el que es fa és assignar a cada element del conjunt un nombre aleatori entre 1 i n, sense repetir cap nombre, i després ordenar el conjunt per aquests nombres. Els algorismes d'ordenació de llistes més ràpids, com *quicksort* o *mergesort*, tenen complexitat $O(n \cdot log(n))$, però com la llista és d'enters podem utilitzar *radix sort*, que té $O(n)$, igual que Fisher-Yates. Aquest mètode té l'inconvenient d'haver de crear i guardar n nombres aleatoris, però també té l'avantatge de que, al contrari que Fisher-Yates, aquest algorisme es pot paral·lelitzar.

3.5.3 Shuffle Morton

Com ja s'ha dit anteriorment, un dels ordres pels punts dels *point clouds* que s'estudiaran és el que s'anomena *shuffle morton*, explicat a l'algorisme 6, que consisteix a ordenar els punts del *point cloud* amb el codi de Morton i, un cop fet això, separar els punts en trossos d'una mida determinada i fer una operació de *shuffle* amb aquests trossos, deixant l'ordre intern de cada tros intacte. Amb això el que es pretén és mantenir una localitat

alta entre els trossos, però alhora fer que la càrrega de processar els punts es reparteixi entre els fils de la GPU de forma més eficient.

Quan es fa la rasterització de la geometria de l'escena, el *viewport*, o espai de projecció, es divideix en una sèrie de "rajoles", i totes les primitives es classifiquen segons en quina rajola estan situades. En arquitectures de NVIDIA, cada rajola està assignada a un grup de processadors de gràfics (*Graphics Processing Cluster* o GPC a partir d'ara), que és un conjunt de *warps* que té accés exclusiu al *framebuffer* d'aquesta part del *viewport*. Aquesta assignació és estàtica, és a dir, està decidida per la construcció de la GPU i no es pot canviar. Aquests GPC estan distribuïts en patrons al llarg del *viewport* per repartir la càrrega de treball el màxim possible. Encara així, aquests patrons no poden contrarestar l'efecte de tenir molts punts agrupats junts, ja que tots caurien al mateix GPC. És per això que s'utilitza aquest mètode *shuffle Morton*, ja que preserva una localitat entre els punts, però tanta localitat que el treball es reparteix de forma ineficient. Això es deu al fet que, a l'ordre Morton original, es té que molts punts consecutius van a parar tots al mateix GPC (ja que hi ha alta localitat), i això fa que només s'executi aquest GPC, afectant negativament a la paral·lelització del treball, però quan es fa el *shuffle*, quedarà una seqüència de trossos on es tindran diversos GPCs intercalats, i per tant la feina es farà en paral·lel, augmentant el rendiment total.

Algorithm 6 Algorisme de Shuffle Morton

```
Input: Point cloud:pc
Output: Point cloud ordenada:pcShuffleMorton
function MAIN(pc)
    punts = pc.getPoints()
    puntsInt = transformToInt(punts)
    codis = calcMortonCode(puntsInt)
    indexos = indexThatSortList(codis)
    puntsSorted = order(punts, indexos)
    batchSize = 128
    pointBatches = dividePointsInBatches(puntsSorted, batchSize)
    puntsShuffled = shuffle(pointBatches)
    pcShuffleMorton = createPointCloud(puntsShuffled)
    return pcShuffleMorton
end function
```

Una vegada introduïda la notació i els algorismes principals a estudiar, en el capítol següent s'explicaran el disseny i la implementació dels algorismes estudiats i del programa visualitzador de *point clouds*.

Capítol 4

Desenvolupament del software

Un dels objectius per aquest treball és desenvolupar un programa que sigui capaç de comparar les visualitzacions de *point clouds* i analitzar els seus costos utilitzant els mètodes estudiats. En aquest capítol, es detalla el disseny i desenvolupament d'aquest programa i el seu funcionament, analitzant cada fase de la seva *pipeline* gràfica.

4.1 Arquitectura del Sistema

4.1.1 Pipeline gràfica

El procés de visualitzar dades en 3D necessita una *pipeline* ben definida, amb etapes com el modelatge, la definició de les estratègies de *rendering*, l'enviament de les dades a la GPU i la realització de l'estratègia de *rendering*. S'han dissenyat i implementat dues *pipelines* diferents: una d'elles seguint l'estructura del *pipeline* gràfic explicat a la secció 1.1, i l'altra basada en la utilització de *compute shaders* i inspirada per aquest article[46].

La primera *pipeline* implementada està il·lustrada a la figura 4.1. Les dues primeres etapes es realitzen a la CPU. A la primera etapa: **1) Inicialització, configuració i lectura de dades**, s'estableix el que serà el bucle de *rendering* de l'aplicació, es configura el context de l'aplicació, es crea el món, l'escena, la càmera... I s'inicialitza tot. També és en aquest pas on es llegirà i es carregarà el *point cloud*. En el segon pas, **2) Enviament de dades a la GPU**, es defineixen quines dades es volen enviar a la GPU, que en el cas dels *point clouds* són les posicions dels punts i els seus colors (en aquest cas, el *point cloud* $\mathbf{P} = (P, A)$ té A igual a l'espai RGB). Per això creem un VAO (*Vertex Array Object*) i dos *buffers*, que serveixen de connexió entre la CPU i la GPU. Les següents dues etapes es fan a la GPU: **3) Vertex Shader**, que en aquesta *pipeline* calcula la posició de món i el color de cada punt, i **4) Fragment Shader**, que decideix quin és el color amb què es pintarà cada píxel. A més a més, és en aquesta fase on es fa el Z-Test, que comprova si el punt és visible o no.

La segona *pipeline* implementada, il·lustrada a la figura 4.2, és bastant més diferent. Les primeres dues etapes són exactament les mateixes que a la *pipeline* 1, amb la única excepció de que, quan es passen les dades del *point cloud* a la GPU, en comptes d'un

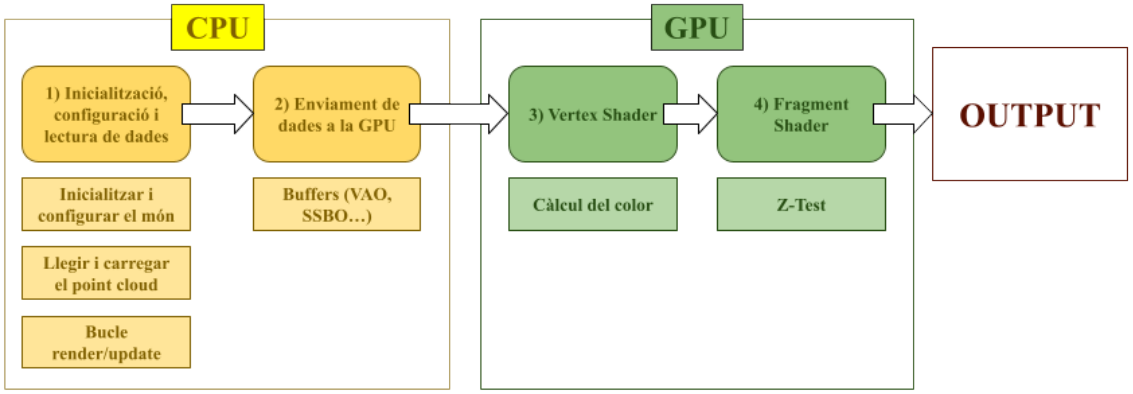


Figura 4.1: Esquema de la *pipeline* gràfica 1.

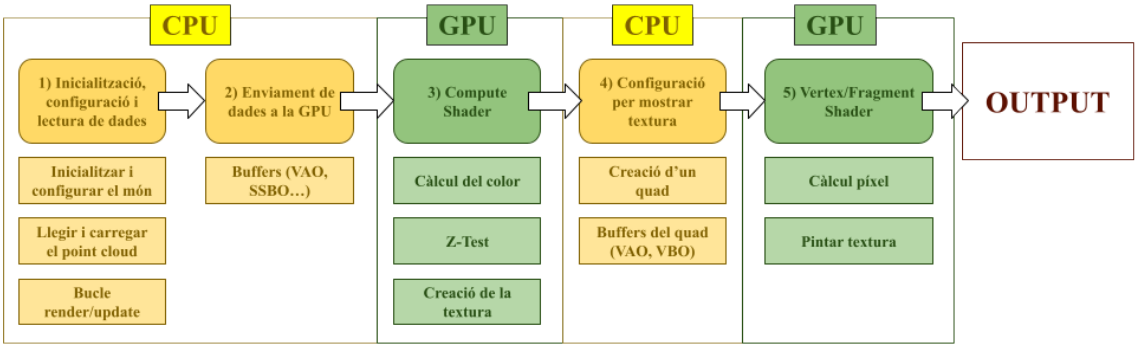


Figura 4.2: Esquema de la *pipeline* gràfica 2, que utilitza un *compute shader* per a fer càlculs de color i profunditat.

VAO s'utilitza un SSBO (*Shader Storage Buffer Object*), i es fan a la CPU. És a la tercera etapa **3) Compute Shader** on trobem la primera diferència. Aquest *compute shader*, la implementació del qual està detallada a la secció 4.4.2, realitza el Z-Test, fa el càlcul del color i guarda les dades de profunditat i color en un *buffer* específic, com si fos una textura 2D que s'ha "aplanat"per fer-la en una array 1D. Aquest *compute shader* s'executa a la GPU, i formaria part de la fase d'aplicació de la *pipeline* gràfica de la secció 1.1. Després es passa a la fase **4) Configuració per mostrar textura**, on bàsicament creem un quad (dos triangles) que ocupen la pantalla sencera, s'envia el quad a la GPU utilitzant un VAO i un VBO (*Vertex Buffer Object*). Per acabar està l'etapa **5) Vertex/Fragment Shader**, que són un *vertex shader* i un *fragment shader* molt bàsics que calculen la posició i el ID de cada píxel, agafen la posició del *buffer* de "textura"que toca, extreuen el color i pinten el píxel.

4.1.2 Inicialització, configuració i lectura de dades

A les dues *pipelines* gràfiques que s'han creat per al projecte, el primer pas consisteix a muntar i configurar tots els elements necessaris per a poder visualitzar una escena. En aquesta secció s'expliquen quins són aquests objectes i com s'han configurat:

- **Càmera**: Per a la càmera definim la seva posició $L = (L_x, L_y, L_z)$ (en coordenades de món), el punt cap a on mira $D = (D_x, D_y, D_z)$, un vector z-up igual al vector de món per la coordenada que mira amunt $\vec{V}$, els angles de rotació (α, β, γ), el FOV (*Field of View*) i la relació d'aspecte s. Amb aquestes dades es poden crear les matrius de vista **ViewMatrix** (3.3.6) i de projecció **ProjectionMatrix** (3.3.13).
- **Point cloud**: Es llegeix i es crea el *point cloud* **P**. En aquest projecte, el *point cloud* ha de ser un arxiu de tipus .ply (Polygon File Format), que tingui una llista de punts amb les propietats de que tots els punts tinguin coordenades en float i color rgb on cada color sigui un char, perquè així ocupen 8 bits. En cas de tenir més propietats al punts, s'hauria de fer algun tipus de preprocessament per a deixar només les propietats esmentades.
- **Shaders i les seves variables**: Aquí es creen els *shaders* i es defineixen les variables que s'encarregaran de determinar quin *shader* (i per tant, quina estratègia de renderitzat) s'està utilitzant.

Després d'això, s'inicialitza el bucle infinit (o fins que es tanqui la finestra) de l'aplicació. En aquest bucle tenim dues fases que s'alternen contínuament: actualització i renderització.

A la fase d'actualització, s'actualitzen les variables del programa. Es comença pels inputs del ratolí, com els clics o el moviment. Després es canvien les matrius de vista i projecció d'acord amb aquests inputs. En aquesta fase també es calcula el moviment de la càmera.

A la fase de renderització, primer es renderitza la interfície gràfica de l'usuari. Després es renderitza el *point cloud*, enviant-lo a la GPU per a que els *shaders* facin tots els càlculs necessaris.

4.2 Implementació del Codi Morton

La implementació de l'algorisme que ordena els punts del *point cloud* seguint el codi de Morton s'ha realitzat en un script de Python, de forma externa al programa principal. L'algorisme utilitzat segueix el model explicat a la secció 3.4.2. A continuació s'explica com s'han implementat les diferents funcions de l'algorisme:

- **1. Transformació dels punts**: Per a realitzar aquesta etapa es defineix una funció que agafa una llista de punts amb coordenades en float $p = (x, y, z)$ i els escala per a que tinguin forma $p' = (x', y', z')$, on $x', y', z' \in \{0, 1, ..., 2^n - 1\}$, on n és el nombre de bits utilitzats pel codi Morton. S'utilitzen $n = 10$ bits per coordenada, per tal de poder emmagatzemar les tres coordenades en un únic nombre enter de 32 bits. Per a fer aquesta transformació s'utilitza, per a cada coordenada, la següent fórmula:

$$x' = \lfloor \frac{x - x_{min}}{x_{max}} \cdot (2^n - 1) \rfloor \tag{4.2.1}$$

  On x_{min}, x_{max} són el mínim i màxim de la coordenada x entre tots els punts ($x_{min}, y_{min}, z_{min}$ no necessàriament han de ser al mateix punt). La primera part

de la fórmula garanteix un nombre entre 0 i 1, en multiplicar per $2^n - 1$ s'aconsegueix el que es vol. Com que d'aquesta multiplicació pot sortir un nombre no enter, s'agafa la part entera del resultat (en realitat es fa un typecast de float a int, però matemàticament és agafar la part entera).

- **2. Càlcul del codi Morton**: Per aquesta part s'han creat dues funcions. La primera funció rep un nombre enter n, agafa els 10 bits menys significatius i realitza una sèrie d'operacions amb els bits de tal manera que introdueix dos zeros entre cada parell de bits. A continuació es detalla quines operacions són, i posem com a exemple $n = 894 = 0x0000037E$ en hexadecimal:
  - $n = n$ & $0x3FF$. Aquesta operació selecciona els 10 bits menys significatius. En teoria el nombre introduït no hauria de tenir més de 10 bits, ja que al pas anterior ens assegurem de que tinguin 10 bits, però es seleccionen igualment, per robustesa. A l'exemple, aquests 10 bits són: 1101111110, per tant, voldríem com a resultat $0000|1001|0000|0100|1001|0010|0100|1000 = 0x09049248$ en hexadecimal. S'observa que posar dos zeros entre cada bit dona un nombre de 28 bits, per arribar a 32 bits els 4 més signifactius estan sempre a 0.
  - $n = (n|n << 16)\&0x030000FF$. Aquesta operació mou els bits de n 16 posicions cap a l'esquerra i els suma a n, i després es seleccionen els 8 bits menys significatius i els bits 25 i 26. Exemple: $n = 0x0000037E$, $(n|n << 16) = 0x037E037E$, $(n|n << 16)\&0x030000FF = 0x0300007E$.
  - $n = (n|n << 8)\&0x0300F00F$. Aquesta operació mou els bits de n 8 posicions cap a l'esquerra i els suma a n, i després es seleccionen els 4 bits menys significatius, els bits 13-16 i els bits 25 i 26. Exemple: $n == 0x0300007E$, $(n|n << 8) = 0x03007E7E$, $(n|n << 8)\&0x0300F00F = 0x0300700E$.
  - $n = (n|n << 4)\&0x030C30C3$. Aquesta operació mou els bits de n 4 posicions cap a l'esquerra i els suma a n, i després es seleccionen els bits 1, 2, 7, 8, 13, 16, 18, 20, 25 i 26. Exemple: $n = 0x0300700E$, $(n|n << 4) = 0x330770EE$, $(n|n << 4)\&0x030C30C3 = 0x030430C2$.
  - $n = (n|n << 2)\&0x09249249$. Aquesta operació mou els bits de n 2 posicions cap a l'esquerra i els suma a n, i després es seleccionen els bits 1, 4, 7, 10, 13, 16, 19, 22, 25, 28. Exemple: $n = 0x030430C2$, $(n|n << 2) = 0x0F14F3CA$, $(n|n << 2)\&0x09249249 = 0x09049248$, que és el resultat que esperàvem.

  La segona funció rep tres nombres x, y, z, fa la primera funció per a cada nombre, desplaça el resultat de y un bit a l'esquerra, el resultat de x dos bits a l'esquerra i els suma, de tal manera que queden els bits de les tres coordenades intercalats, que és el que es vol per obtenir el codi de Morton.

- **3. Ordenació dels punts**: Per a implementar aquesta part s'ha utilitzat la funció de Python np.argsort, que ens retorna una llista amb els índexs que ordenen una llista, i s'utilitza per ordenar els punts del *point cloud*.

- **4. Creació del point cloud**: Es crea un nou objecte *point cloud* amb les dades ordenades i es guarda per poder utilitzar-ho al programa principal.

Per a fer les operacions de *shuffle* necessàries per els ordres *shuffle* i *shuffle Morton*, s'ha utilitzat la funció random.shuffle de Python, que implementa l'algorisme de Fisher-Yates, explicat a la secció 3.5.2.

4.3 Càlcul Color

A l'hora de decidir de quin color pintar els punts, existeixen moltes estratègies possibles que es poden utilitzar: Phong, Blinn-Phong o Gouraud. Cada tècnica té les avantatges i els seus inconvenients, i totes tenen un cert cost computacional associat.

En el projecte ens centrem única i exclusivament en el rendiment del programa, per tant, simplement agafem el color del punt més proper per a cada píxel i el pintem.

4.4 Renderització i Visualització

4.4.1 Pipeline clàssica amb GLSL

En aquesta secció s'explica en detall el funcionament de la *pipeline* gràfica clàssica de GLSL que s'ha implementat. Aquesta *pipeline* comença amb els passos detallats a la secció 4.1.2 per inicialitzar el programa. Després s'envia el *point cloud* a la GPU, i per a fer això es crea un VAO i dos *buffers*, un per a guardar les posicions del punts i un altre per guardar els seus colors. A continuació es crida a la funció de *draw*, que s'encarrega de dibuixar el *point cloud* utilitzant els *shaders* corresponents. Per a realitzar aquesta tasca es crea la **modelMatrix** (que en aquest cas és la matriu identitat) i es passa a la GPU, juntament amb la matriu de vista M_{view} i la matriu de projecció M_{proj}, explicades a la secció 3.3, i també l'hem d'indicar quants píxels ha d'ocupar cada punt. Aquest paràmetre s'ha inicialitzat a 1, per a que hi hagi una correspondència entre punts i píxels. Tanmateix, en el cas de que hi haguessin molts forats a la imatge resultant es podria fer més gran per mitigar l'impacte dels forats, però no massa gran, ja que es podria perdre resolució de la imatge. A continuació a les figures 4.3 i 4.4 veiem els *shaders* utilitzats en aquesta *pipeline*.

```
#version 330 core

layout (location = 0) in vec4 vPosition;
layout (location = 1) in vec4 vColor;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

out vec4 color;

void main() {
    gl_Position = projectionMatrix*viewMatrix*modelMatrix * vPosition;
    color = vColor;
}
```

Figura 4.3: Vertex shader

```
#version 330

in vec4 color;
out vec4 colorOut;

void main()
{
  colorOut = color;
}
```

Figura 4.4: Fragment shader

En quant als *shaders*, primer tots els punts passen pel *vertex shader*, que calcula la posició del punt dins del món fent la multiplicació de matrius model, vista i projecció, i

passa aquest valor i el color del punt al *fragment shader*. El *fragment shader*, per la seva part, l'únic que fa el passar el color del punt a pantalla. És després d'aquest *shader* on s'efectua el Z-Test que implementa OpenGL, tot i que en versions més recents de GL es fa el Z-Test abans d'entrar al *fragment shader*. Aquest Z-Test el que fa és guardar el valor z (profunditat) del fragment més proper i comprovar si el nou fragment està més a prop o no.

Respecte als diferents ordres de punts que s'han estudiat en aquest projecte, l'ordre de Morton permet, com hem dit anteriorment, desactivar el Z-Testing per un cert punt de vista i una situació concreta sense cap canvi en la imatge final. Això dona rendiment addicional, ja que s'eviten les operacions d'escriure al *buffer* i comparar. En els altres ordres treure el Z-Testing (i per tant el Z-Buffer) també milloraria el rendiment, però la imatge no es veuria correctament.

4.4.2 Tècniques d'Optimització en Compute Shaders i GLSL

En aquesta secció detallarem el funcionament de la *pipeline* gràfica basada en *compute shaders* que s'ha implementat en aquest treball. Aquesta *pipeline* comença amb els passos detallats a la secció 4.1.2 per inicialitzar el programa. Després s'envia el *point cloud* a la GPU, i per a fer això es creen dos *buffers* SSBO, un per a guardar les posicions del punts i un altre per guardar els seus colors. A continuació es crida a una funció que realitza el pas de *compute shader*, il·lustrat a les figures 4.6 i 4.5. Per això, primer s'ha d'enviar la mida del *viewport* i la matriu de model (que és la identitat en aquest cas) i cal inicialitzar el *buffer* que guardarà els punts com a "textura". Després d'això s'executa el *compute shader*, que fa les següents operacions:

- **Clipping**: El *shader* calcula a quina posició dels *buffers* ha de llegir i si es passa de la longitud del *buffer* es retorna. Després es calculen les coordenades de clip fent la multiplicació dels matrius de model (un altre cop és la identitat), vista (M_{view}) i projecció (M_{proj}) amb les coordenades del punt, i si la coordenada t és menor o igual a 0 es retorna, perquè llavors el punt està darrere de la càmera. Després calcula les NDC (*Normalized Device Coordinates*), i si el valor absolut de la coordenada z és més gran que 1 significa que el punt està fora de la pantalla, i també es retorna.
- **Creació del punt**: En aquest pas es crea un vector de 2 ints de 32 bits cadascun, i es posa la profunditat del punt en una coordenada i el color rgb en l'altra. Això es fa per a poder aplicar el següent pas, que és on està l'optimització del *shader*. Aquí també calculem el ID del píxel en el qual està el punt.
- **AtomicMin**: Per a calcular el color i la profunditat alhora es vol codificar les dues variables en un mateix int de 64 bits, on els 24 bits menys significatius són el color rgb, i els següents 32 bits la profunditat, per tal de poder fer operacions atòmiques de comparació per agafar la profunditat més petita (`atomicMin`). Tanmateix, la GPU utilitzada no sembla capaç de fer operacions atòmiques amb ints de 64 bits, per tant s'ha hagut d'implementar aquesta idea amb un vector de 2 ints de 32 bits, on es guarda la profunditat a una coordenada i el color a l'altra. També s'ha hagut de crear una funció per a poder comparar els vectors com si fossin ints de 64 bits. En aquest pas s'agafa el int creat al pas anterior i es compara amb el punt que està al *framebuffer* dedicat (s'utilitza el fet que se sap el ID del píxel per saber amb quina

```
void main() {

    uint i = gl_GlobalInvocationID.x +
             gl_GlobalInvocationID.y * gl_NumWorkGroups.x * gl_WorkGroupSize.x;
    if (i >= positions.length()) return;

    vec4 pos = projectionMatrix*viewMatrix*modelMatrix * positions[i];
    if (pos.w <= 0.0) return; // behind camera

    vec3 ndc = pos.xyz / pos.w;
    if (abs(ndc.x) > 1.0 || abs(ndc.y) > 1.0) return; // off-screen

    // convert to pixel coordinates
    uint x = uint((ndc.x * 0.5 + 0.5) * float(imageWidth));
    uint y = uint((ndc.y * 0.5 + 0.5) * float(imageHeight));
    if (x >= imageWidth || y >= imageHeight) return;

    uint pixelID = x + y * imageWidth;
    uint depthKey = floatBitsToInt(ndc.z);
    uint rgb = packRGB(colors[i].rgb);

    uvec2 point = uvec2(rgb, depthKey);
    atomicMin64(pixelID, point);
}
```

Figura 4.5: Funció main del *compute shader*

posició del *framebuffer* comparar), i si té una profunditat més petita s'intercanvien. Així s'assegura que es té per a cada píxel el punt més proper i el seu color al mateix temps.

Un cop s'ha executat el *compute shader*, es crida a una altra funció, que canvia el *shader* actiu per un *shader* molt simple que s'utilitza per mostrar la imatge per pantalla. Tot seguit aquesta funció crea un objecte quad, compost de dos triangles, que ocupa tot el *viewport*, i també crea dos *buffers* VAO i VBO per enviar les dades del quad a la GPU. Després es dibuixa el quad, per a fer això s'utilitza el parell *vertex/fragment shader* següent. El *vertex shader* només calcula la posició de cada punt. El *fragment shader* calcula el ID del píxel en què estem, accedeix al *buffer* on ha escrit el *compute shader*, treu el punt que toca, extreu el seu color i pinta el píxel. Un cop s'ha fet això es torna a activar el *compute shader* per a repetir el procés al següent fotograma. Aquest parell *vertex/fragment shader* es mostra a les figures 4.7 i 4.8.

Respecte als diferents ordres dels punts, un altre cop l'únic que podria canviar l'eficiència d'aquesta pipeline és el codi de Morton, ja que podria permetre evitar el Z-Test, en els casos comentats en el capítol 3.

```glsl
#version 450

layout(local_size_x = 16, local_size_y = 16) in;

// Binding 0 = positions
layout(std430, binding = 0) readonly buffer Positions { vec4 positions[]; };
// Binding 1 = colors
layout(std430, binding = 1) readonly buffer Colors { vec4 colors[]; };
// Output framebuffer (emulated 64-bit with 2x32-bit)
layout(std430, binding = 2) buffer Framebuffer { uvec2 framebuffer[]; };

uniform uint imageWidth;
uniform uint imageHeight;

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;

uint packRGB(vec3 c) {
    uvec3 rgb = uvec3(clamp(c * 255.0, 0.0, 255.0));
    return (rgb.r << 16) | (rgb.g << 8) | rgb.b;
}

// Compare two uvec2 "64-bit" values
bool less64(uvec2 a, uvec2 b) {
    return (a.y < b.y) || (a.y == b.y && a.x < b.x);
}

// Emulated atomicMin for 64-bit (two 32-bit atomics)
void atomicMin64(uint idx, uvec2 newVal)
{
    for (;;) {
        uvec2 old = framebuffer[idx];
        if (!less64(newVal, old)) return;

        uint oldLow = old.x;
        uint swapped = atomicCompSwap(framebuffer[idx].x, oldLow, newVal.x);
        if (swapped == oldLow) {
            uint oldHigh = old.y;
            uint swappedHigh = atomicCompSwap(framebuffer[idx].y, oldHigh, newVal.y);
            if (swappedHigh == oldHigh)
                return;
        }
    }
}
```

Figura 4.6: Funcions auxiliars del *compute shader*

```
#version 450
layout(location = 0)
in vec2 pos;
void main() {
    gl_Position = vec4(pos, 0.0, 1.0);
}
```

Figura 4.7: *Vertex shader*

```
#version 450
layout(std430, binding = 2) readonly buffer Framebuffer { uvec2 fb[]; };
uniform uint imageWidth;

in vec2 uv;
out vec4 fragColor;

vec3 unpackRGB(uint pack) {
    return vec3(
        float((pack >> 16) & 0xFFu),
        float((pack >> 8) & 0xFFu),
        float(pack & 0xFFu)
    ) / 255.0;
}

void main() {
    uint x = uint(gl_FragCoord.x);
    uint y = uint(gl_FragCoord.y);
    uint pixelID = x + y * imageWidth;

    // fb[pixelID].x = rgb, fb[pixelID].y = depth
    uvec2 val = fb[pixelID];
    uint rgb = val.x;   // read packed color
    fragColor = vec4(unpackRGB(rgb), 1.0);
}
```

Figura 4.8: *Fragment shader*

Capítol 5

Resultats Experimentals

En aquest capítol s'explica quin és el *hardware* utilitzat per a fer les simulacions, quins *point clouds* s'han utilitzat i quines són les seves característiques i quins mètodes s'avaluaran i quines mètriques s'utilitzaran per avaluar-los. A més, es mostren i comenten els resultats obtinguts.

5.1 Hardware utilitzat i Programari desenvolupat

Totes les proves i experiments realitzats en aquesta secció han estat fetes en un ordinador concret. Evidentment les característiques poden afectar al rendiment del nostre programa, és per això que aquí donarem els detalls de la nostra màquina. Les especificacions de l'ordinador utilitzat a les proves són:

- **Sistema operatiu**: Ubuntu 24.04.3 LTS (utilitzem WSL2 en un ordinador amb Windows 10 Home).
- **Processador**: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz
- **RAM instalada**: 16,0 GB
- **Targeta gràfica**: NVIDIA GeForce GTX 1050 (4GB)
- **Memòria de GPU**: 11,9 GB
- **Memòria de GPU dedicada**: 4,0 GB
- **Memòria de GPU compartida**: 7,9 GB
- **Versió del controlador**: 32.0.15.8129

El projecte que s'ha usat està penjat en el següent repositori de Github: [16]. En el README.md estan les instruccions per instal·lar el programa.

5.2 Metodologia

En aquesta secció, es comença detallant quins són els *datasets* que s'han decidit utilitzar, i es continua dient quines mètriques s'han escollit per a valorar quins ordres i quines estratègies de renderitzat són millors.

5.2.1 Descripció dels datasets

Els *datasets* que s'han utilitzat en aquest treball són els següents:

- **Crani de cabra**: Aquest *point cloud*, extret de Artec 3D [2], representa un crani de cabra. Està en format de .ply *little endian*, i té com a atributs les posicions dels punts (coordenades x, y, z) com a floats i el seu color rgb com a uchar. Aquest *point cloud* conté 983599 punts.
- **Gerro amb textura**: Aquest *point cloud*, també extret de Artec 3D [2], representa un gerro amb dibuixos. Està en format .ply *little endian*, i té les coordenades x, y, z expressades com a floats i els colors rgb com a uchar. Aquest *point cloud* conté 412755 punts.
- **Manuscrit**: Aquest *point cloud*, extret del repositori d'escanejos 3D de Stanford [4], representa una pàgina d'un llibre medieval. Està en format .ply *big endian*, i té les coordenades x, y, z com a floats, i els colors rgb coma uchar. Aquest *point cloud* conté 2155617 punts.
- **Avió**: Aquest *point cloud*, extret del TFG del Gerard Perelló [39], representa un avió. S'ha modificat aquest *point cloud* per a que tingui el format que necessitem, així que s'han tret les labels i s'ha fet que tots els punts siguin blancs. Aquest *point cloud* només té 2048 punts, i l'utilitzarem per a comparar amb el treball del Gerard i veure quin mètode és més eficient.

A continuació s'inclou la taula 5.1, que fa un resum dels núvols de punts.

Point Cloud	Nombre de punts
Cabra	983599
Gerro	412755
Manuscrit	2155617
Avio	2048

Taula 5.1: Point clouds i el seu nombre de punts.

5.2.2 Mètriques utilitzades

En aquest projecte, com que s'està avaluant el rendiment del programa, s'ha usat com a mètrica els fotogrames per segon (fps), on més fps és millor, ja que representa una visualització més fluida, i per veure les diferències més concretes s'han mesurat els ms necessaris per construir un *frame*. També s'analitza si les imatges generades són correctes, és a dir, no tenen forats ni artefactes visuals.

5.2.3 Mètodes avaluats

Els mètodes de renderització avaluats són:

- **GL_Points sense Z-Test**: Aquest és el mètode implementat a la *pipeline* clàssica de la secció 4.4.1, utilitzant la primitiva de punts *GL_Points*. La variable de la mida dels punts *glPointSize* té el valor per defecte. Aquest mètode no implementa el Z-Test
- **GL_Points amb Z-Test**: Igual que el mètode anterior, però aquí activem el Z-Test que fa OpenGL amb *glDepthTest*.
- **Compute Shader amb early Z-Test**: Aquest és el mètode implementat a la *pipeline* descrita en la secció 4.4.2, utilitzant un *compute shader* que fa operacions d'`atomicMin` per a agafar el punt amb menys profunditat. També implementa un early Z-Test per a no executar operacions `atomicMin` quan no faci falta.
- **Compute Shader sense Z-Test**: versió diferent del *compute shader* on no es comprova la profunditat dels punts i es posen al *framebuffer* a mesura que es van trobant. Equivalent al primer mètode però amb *compute shader*.

I els ordres pels punts avaluats són:

- **Original**: Els punts es queden en el mateix ordre en que vam agafar el *point cloud*. Alguns *point clouds* ja estan ordenats amb algun ordre específic, i d'altres van per ordre d'escaneig o barrejats per simular estructures de nivell de detall.
- **Codi Morton**: El *point cloud* s'ordena usant un codi de Morton de 10 bits, que és equivalent a fer un DFS (*Depth-First Search*) d'un octree amb 10 nivells.
- **Shuffle**: Els punts es barregen seguint l'algorisme de Fisher-Yates.
- **Morton Shuffle**: Els punts s'ordenen exactament igual que al codi Morton, després es separen en grups de 128 punts i es barregen aquests grups usant l'algorisme de Fisher-Yates, preservant l'estructura interna dels grups. Com ja hem dit anteriorment, això té com a objectiu mantenir una localitat entre punts, i que alhora el treball de la GPU es distribueixi de forma més equitativa.

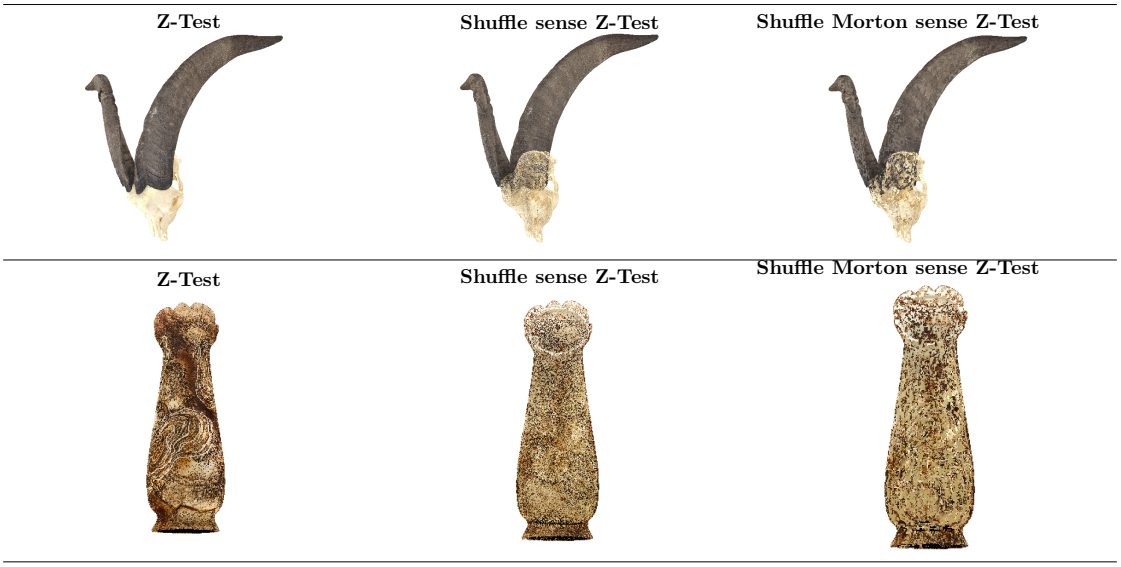
5.3 Resultats

En aquesta secció s'avaluaran les estratègies d'optimització de visualització de *point clouds* que s'han estudiat i es compararan entre elles. Cal dir que en aquesta secció només es comentaran els resultats de la versió de Linux, els resultats de la versió de Windows es troben a l'apèndix A.

Per a poder saber el nombre de FPS de cada mètode i cada ordre, el programa mostra un comptador on mostra el FPS actuals. S'ha fet que addicionalment el programa vagi imprimint aquests nombres per consola, i s'ha creat un script de bash que executa el

programa durant 60 segons, llegeix totes les lectures de FPS i fa la mitjana. Després per aconseguir els mil·lisegons s'ha dividit 1000 pel nombre de FPS. Addicionalment, a totes les visualitzacions, la càmera s'ha col·locat en l'octant que fa que el codi Morton no necessiti Z-Test.

Primer, s'analitza la qualitat de les imatges generades per cada *shader*. Les versions dels *shaders* que no implementen Z-Test presenten greus distorsions a la visualització de les imatges, que queden molt diferent de com s'haurien de veure, com es mostra a la taula 5.2. Aquestes distorsions canvien segons el tipus d'ordre, sent més notables al ordres *Shuffle* i *Shuffle Morton*, i segons el *point cloud*, sent més comunes a la cabra i el gerro. Això és esperable, ja que el Z-Test s'encarrega de que es mostri el punt més proper a la càmera per a cada píxel, i els ordres que barregen els punts fan que el punt que es mostri a cada píxel sigui aleatori. Notem que aquesta distorsió no existeix quan s'observen les versions ordenades amb codi Morton des de determinats angles. També observem que gairebé no hi ha pertorbacions o errors als *point clouds* de l'avió ni al manuscrit; al primer exemple perquè no hi han suficients punts com per a hi hagi diversos punts al mateix píxel (i perquè tots els punts els hem fet del mateix color), i al segon exemple perquè el manuscrit és un *point cloud* pla, per tant no té molta profunditat, el que significa que hi ha pocs punts que es superposin i puguin provocar distorsions.



Taula 5.2: Visualització dels *point clouds* de la cabra i el gerro comparant la imatge quan hi ha Z-Test i quan no hi ha. Es veuen errors greus a les imatges sense Z-Test.

La taula 5.4 mostra els resultats d'executar totes les estratègies en tots els ordres possibles per a diferents *datasets*, des de 2048 fins a 2 milions de punts, juntament amb imatges dels *point clouds*. Els resultats mostren de forma clara que els mètodes basats en *compute shaders* són molt més ràpids que simplement projectar els punts amb *GL_Points*, aconseguint entre 2 i 4 vegades més FPS.

Una notable excepció és el *point cloud* de l'avió, on és el *shader* tradicional el que és més ràpid. Això es deu al fet que aquest *point cloud* és molt petit, i el *compute shader* té un cost inicial alt, però escala més lent amb el nombre de punts, i de fet es pot veure això en les gràfiques, on el *compute shader* és un 41% més lent a l'avió, un 110% més ràpid al gerro, un 220% més ràpid a la cabra i un 275% més ràpid a la pàgina comparat amb

GL_Points. Observem també que les versions dels *shaders* que no implementen Z-Test són lleugerament més ràpides, independentment del *point cloud* i de l'ordre dels punts, el qual és esperable, ja que no tenir Z-Test implica menys operacions realitzades i per tant més rapidesa.

En quant als ordres dels punts, també es pot veure el seu impacte a la taula 5.4. Al *point cloud* de l'avió observem que l'ordre dels punts no té gaire importància, això és degut a que és un *point cloud* molt petit, i al tenir pocs punts l'impacte de l'ordre dels punts és negligible. Al altres tres *point clouds*, però, veiem que l'ordre original i l'ordre de Morton acostumen a tenir el mateix rendiment independentment del *shader*, i que *shuffle* té un rendiment lleugerament inferior, mentre que *Shuffle Morton* és lleugerament més ràpid que els altres ordres.

S'observa que l'ordre *Shuffle Morton* és més eficient que l'ordre Morton. Això es deu en part a que, com ja s'ha dit anteriorment a la secció 3.5.3, la mesura de localitat de *Shuffle Morton* és més alta, fent que la feina es distribueixi als GPC, però suficientment baixa com per no perdre eficiència. A la taula 5.3 tenim la distància mitjana entre punts consecutius, una de les mesures de localitat estudiades a la secció 3.4.3, per a cada *point cloud* amb ordres Morton i *Shuffle Morton*.

En quant a la comparació amb el treball del Gerard Perelló [39], es pot veure il·lustrada a la figura 5.1. Al treball del Gerard s'utilitzava el *ray tracing* com a estratègia de visualització, utilitzant una estructura de *MultiOctree*. És evident que els mètodes estudiats en aquest treball tenen una eficiència superior, ja que no s'utilitza cap algorisme de *ray tracing*.

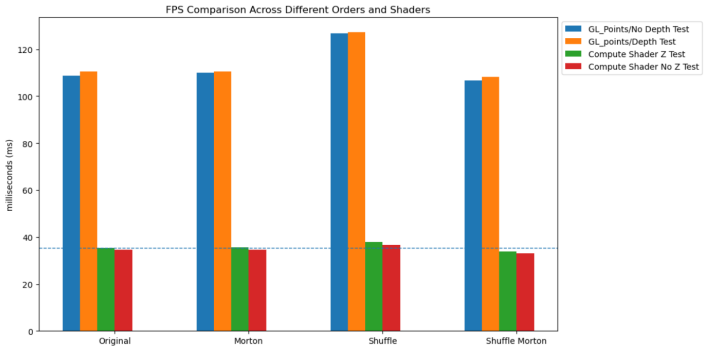
Point Cloud / Ordre dels punts	Localitat mesurada en distància mitjana
Cabra/Morton	0.7516
Cabra/Morton Shuffle	1.7749
Gerro/Morton	2.5488
Gerro/Morton Shuffle	4.7622
Manuscrit/Morton	0.2700
Manuscrit/Morton Shuffle	0.8439
Avio/Morton	0.0486
Avio/Morton Shuffle	0.0535

Taula 5.3: Point clouds i les seves mesures de localitat.

Interpretant els resultats de les simulacions que s'han realitzat, la millor opció és utilitzar *pipelines* basades en *compute shaders* i que implementin early Z-Test per a tractar *point clouds* grans, ja que és el mètode que ens ha donat el major rendiment, sense comptar el *compute shader* sense Z-Test, que ens dona imatges plenes de defectes. També es recomana utilitzar l'ordre *Shuffle Morton*, ja que és lleugerament més eficient que els altres ordres estudiats.

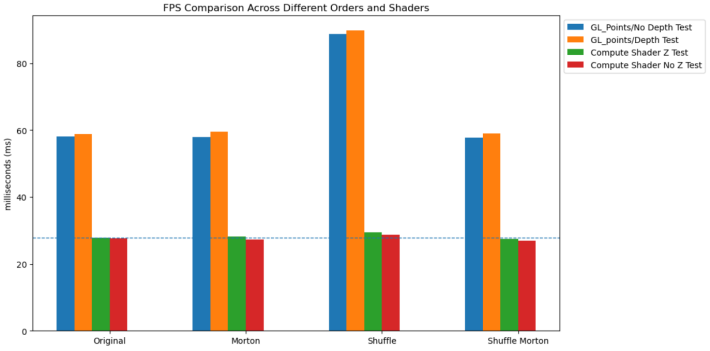
Model: Cabra

Punts: 983599

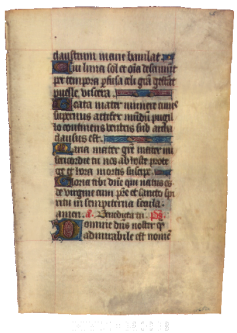


Model: Gerro

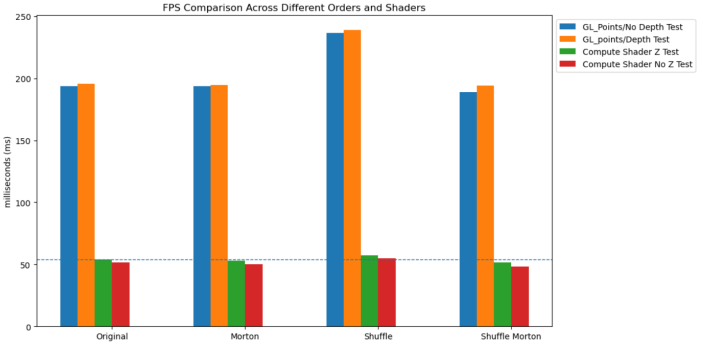
Punts: 412755



Model: Manuscrit

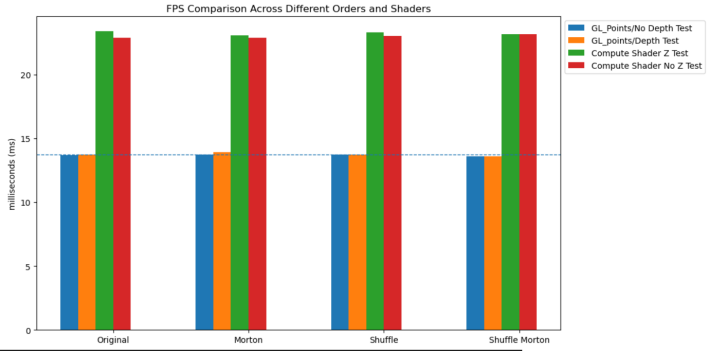


Punts: 2155617



Model: Avió

Punts: 2048



Taula 5.4: Temps per fotograma que obtenim al renderitzar cada *point cloud* amb un mètode concret.

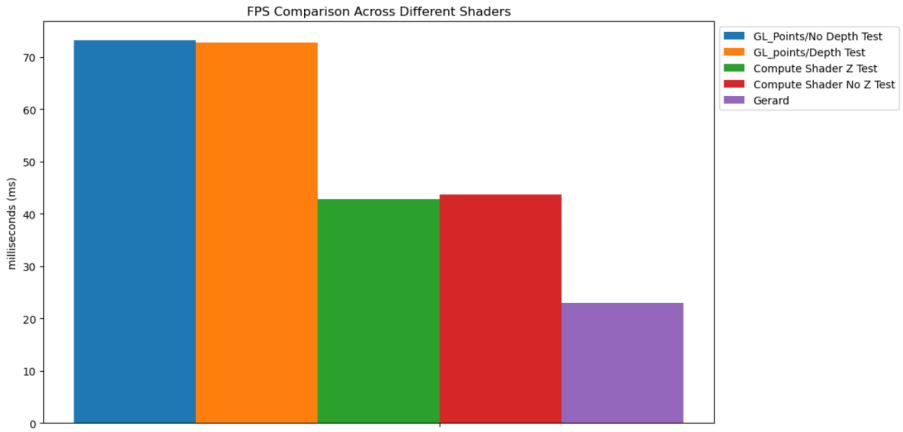


Figura 5.1: Comparació entre el treball del Gerard Perelló i aquest treball.

Capítol 6

Conclusions i Treball Futur

En aquesta secció s'explicaran les conclusions del treball i possibles formes d'estendre'l en el futur, tot això basat en els objectius plantejats a l'introducció i en els resultats que s'han obtingut.

6.1 Conclusions

Després de completar aquest TFG, s'ha aprofundit significativament el coneixement del camp de la visualització de *point clouds*. S'han estudiat i modelat tots els conceptes clau per a la visualització de *point clouds*, que era l'objectiu principal d'aquest treball. Per a fer això, s'han definit tots els conceptes i mètodes des del punt de vista matemàtic. Primer de tot s'ha definit formalment què és un *point cloud*, s'ha aconseguit entendre com funciona una càmera i s'han definit tots els conceptes i les transformacions geomètriques associades. S'ha aprofundit en el món de les corbes per a definir la corba de Lebesgue i s'han estudiat algunes de les seves propietats, i també s'han estudiat les permutacions i els algorismes que les implementen, i s'ha calculat el cost dels algorismes que s'utilitzen.

Al mateix temps s'han utilitzat tots aquests conceptes per a dissenyar i implementar una aplicació que permet visualitzar *point clouds* seguint els mètodes estudiats i definits, utilitzant tècniques d'optimització com ara el early Z-Test o ordenacions de punts com el codi de Morton, de tal manera que el programa pogués tenir el major rendiment. Finalment, s'ha realitzat una anàlisi comparativa dels diferents mètodes i ordres que s'han implementat, mostrant que l'ús de *compute shaders* i d'ordres alternatius com el *Shuffle Morton* mostren potencial de cara a futures investigacions sobre la renderització de *point clouds*.

En resum, aquest treball ha mostrat el funcionament dels mètodes de visualització de *point clouds*, aprofundint en els mètodes basats en *compute shader* i en l'ordre dels punts, i implementant un programa per mostrar tot el que s'ha explicat.

6.2 Treball futur

Una de les possibles millores d'aquest treball seria optimitzar més el codi, i implementar el *compute shader* amb el *framebuffer* de 64 bits, ja que les visualitzacions tenen pocs FPS en haver estat fetes en una targeta gràfica antiga que no té suport per operacions atòmiques de 64 bits. També, en futures investigacions, es podria estudiar l'efecte d'utilitzar altres corbes similars a la corba de Lebesgue, com ara la corba de Hilbert o la de Peano, per a ordenar els punts d'un *point cloud*, i veure quin rendiment tenen en comparació a la corba de Lebesgue. Finalment, seria interessant poder treballar amb *datasets* més grans per estudiar com de bé escalen els *compute shader* amb el nombre de punts.

Apèndix A

Resultats de la versió de Windows

A la taula A.1 es troben els resultats d'executar totes les estratègies en tots els ordres possibles per als diferents *datasets* del treball, des de 2048 fins a 2 milions de punts, juntament amb imatges dels *point clouds*.

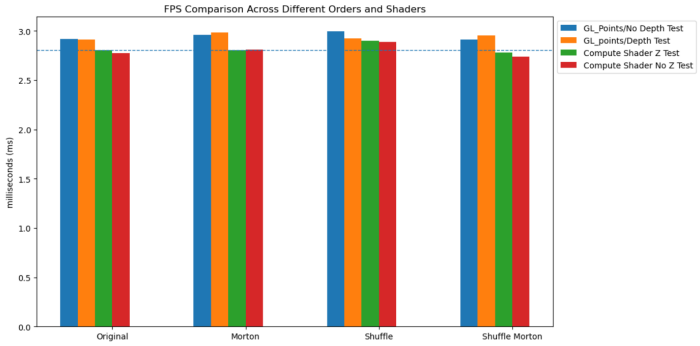
Es pot apreciar clarament que en aquesta versió tots els *point clouds* s'executen molt més ràpid que a la versió de Linux. En l'exemple més extrem, el manuscrit, que podia trigar més de 200 ms per fotograma, només triga una mica més de 3 ms, el que és unes 70 vegades més ràpid. També es pot apreciar que els temps per fotograma no canvien gairebé entre *point clouds*, això significa que per aquesta versió del programa tots aquests *point clouds* són molt petits, i potser amb un *point cloud* més gran podria trigar més temps.

En quant als diferents *shaders*, s'observa que els mètodes basats en *compute shaders* són lleugerament superiors, excepte a l'avió, fet que també passava amb el programa de Linux, i al manuscrit en ordre de *shuffle*. Aquest fet és sorprenent, especialment perquè la versió sense Z-Test, que hauria de ser més ràpida en evitar les operacions de Z-Test, és molt més lenta. S'han mesurat els fps de la combinació manuscrit *shuffle*/*compute shader* diverses vegades per comprovar que no hi havia cap error.

En quant a l'ordre dels punts, en aquests resultats estan tots molt parells, això es deu a que com s'ha dit anteriorment, per aquest programa tots els *point clouds* semblen ser massa petits, però es veu que *shuffle* és una mica més lent que la resta d'ordres, i *Shuffle Morton* és manté igual o una mica superior als altres ordres.

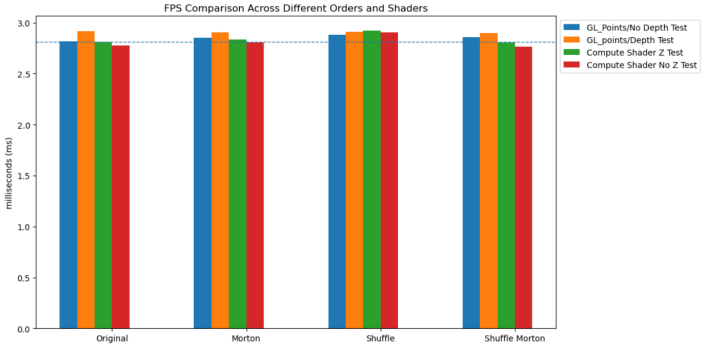
Model: Cabra

Punts: 983599

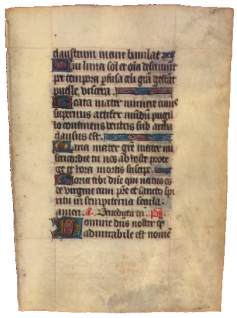


Model: Gerro

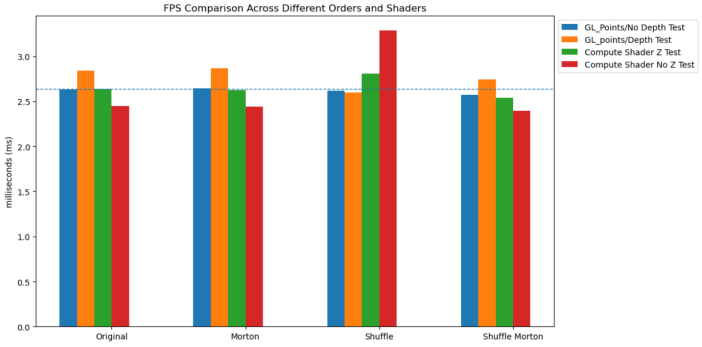
Punts: 412755



Model: Manuscrit

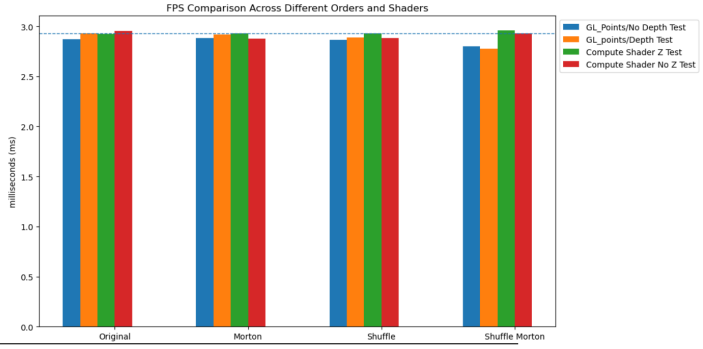


Punts: 2155617



Model: Avió

Punts: 2048



Taula A.1: Temps per fotograma que obtenim al renderitzar cada *point cloud* amb un mètode concret.

Bibliografia

[1] Euler angles. https://en.wikipedia.org/wiki/Euler_angles. Accessed December 2025.

[2] Portable 3d scanners. https://www.artec3d.com/portable-3d-scanners. Accessed December 2025.

[3] Random.py. https://svn.python.org/projects/python/trunk/Lib/random.py.

[4] Stanford 3d scanning repository. https://graphics.stanford.edu/data/3Dscanrep/.

[5] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. CRC Press, Boca Raton, FL, 4 edition, 2018.

[6] Michael Bader. *Space-Filling Curves*. Springer, 2013.

[7] Carmen Cincotti. Homogeneous coordinates, clip space, and ndc. https://carmencincotti.com/2022-05-02/homogeneous-coordinates-clip-space-ndc/, 2022. Accessed: 2025-01-12.

[8] Cmglee. Durstenfeld shuffle. https://commons.wikimedia.org/wiki/File:Durstenfeld_shuffle.svg. Accessed: 2026-01-14.

[9] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22:657, 2003.

[10] Hong-Kai Dai and Hong-Chen Su. On the locality properties of space-filling curves. In *Algorithms and Computation (ISAAC 2003)*, volume 2906 of *Lecture Notes in Computer Science*. Springer, 2003.

[11] Hong-Kai Dai and Hong-Chen Su. Clustering analyses of two-dimensional space-filling curves: Hilbert and z-order curves. *SN Computer Science*, 4(1):8, 2022.

[12] Naila Fauzia, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *Proceedings of CGO*, pages 12–22, 2015.

[13] Wanquan Feng, Jin Li, Hongrui Cai, Xiaonan Luo, and Juyong Zhang. Neural points: Point cloud representation with neural fields, 2021. arXiv:2112.04148.

[14] Greg Foletta. A brief tour of lebesgue curves. https://articles.foletta.org/post/lebesgue-curves/, 2022.

[15] James D. Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2. edition, 1990.

[16] Github. Tfg-victorgonzalez. `https://github.com/VictorGonzalez02/TFG-VictorGonzalez/`, 2025. Accessed: 2026-01-10.

[17] Enrico Gobbetti and Fabio Marton. Layered point clouds: A simple and efficient multiresolution structure. *Computers & Graphics*, 28(6):815–826, 2004.

[18] Christoph Günther, Tim Kanzok, Lars Linsen, and Paul Rosenthal. A gpgpu-based pipeline for accelerated rendering of point clouds. *Journal of WSCG*, 21:153–161, 2013.

[19] Jaber Hasbestan and Inanc Senocak. Binarized octree generation for cartesian adaptive mesh refinement. *Journal of Computational Physics*, 368, 2018.

[20] Henrik. Ray trace diagram. `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg`, 2023. Wikimedia Commons, accessed May 31, 2023.

[21] Qiang Hu, Zhen Zhang, and Wei Hu. Rangeldm: Fast realistic lidar point cloud generation, 2024. arXiv:2403.10094.

[22] John F. Hughes, Andries van Dam, Morgan McGuire, James D. Foley, David F. Sklar, Steven K. Feiner, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Pearson Education, Inc., 3rd edition, 2014.

[23] U. Iqbal, T. Davies, and P. Perez. Recent advances in gpu-accelerated computing for real-time and data-intensive applications. *Sensors*, 24(15):4830, 2024.

[24] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of ACM SIGMOD*, pages 332–342, 1990.

[25] Sriram Kashyap, Rhushabh Goradia, Parag Chaudhuri, and Sharat Chandran. Real-time ray tracing of point-based models. In *Proceedings of I3D*, 2010.

[26] Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, and Markus Steinberger. Effective static bin patterns for sort-middle rendering. *High Performance Graphics*, 2017.

[27] Farzad Khorasani, Rajat Gupta, and Laxmi N. Bhuyan. Efficient warp execution in presence of divergence. In *Proceedings of MICRO*, pages 204–215, 2015.

[28] Dae-Hyun Kim. Evaluation of the performance of gpu global memory coalescing. *Evaluation*, 4(4), 2017.

[29] Petrus E. J. Kivi, Markku J. Mäkitalo, et al. Real-time rendering of point clouds with photorealistic effects: A survey. *IEEE Access*, 10:13151–13173, 2022.

[30] Patrick Labatut, Jean-Philippe Pons, and Renaud Keriven. Efficient multi-view reconstruction of large-scale scenes. In *Proceedings of ICCV*, 2007.

[31] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. In *Proceedings of CVPR*, 2021.

[32] Eric Lengyel. *Foundations of Game Engine Development: Mathematics*. Terathon Software, 2012.

[33] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina, 1985.

[34] Yusen Li, Houman Borouchaki, Hanlin Miao, and Jie Zhang. Implicit function-based 3d reconstruction for point cloud data. *Annals of Mathematics and Physics*, page 202–208, Oct. 2025.

[35] Zhijian Liu, Haotian Tang, Yujun Lin, and Song Han. Point-voxel cnn for efficient 3d deep learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[36] N. Lukač, D. Špelič, and B. Žalik. Hybrid visualization of sparse point-based data using gpgpu. In *Proceedings of CANDAR*, pages 178–184, 2014.

[37] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[38] MithrandirMage. Viewfrustum. `https://commons.wikimedia.org/wiki/File:ViewFrustum.svg`, 2012. Accessed: 2026-01-12.

[39] Gerard Perelló Martínez. Interactive multiple point cloud rendering using gpu-based ray tracing. Master's thesis, 2023.

[40] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH*, pages 335–342, 2000.

[41] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of SIGGRAPH*, pages 371–378, 2001.

[42] Packt Publishing. *Meshing and the Environment*. Packt Publishing, 2018. Accessed: 2026-01-09.

[43] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH*, pages 343–352, 2000.

[44] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Rendering Techniques 2000*, pages 319–328, 2000.

[45] Christoph Schied, Anton Kaplanyan, Chris Wyman, et al. Spatiotemporal variance-guided filtering. In *High Performance Graphics*, 2017.

[46] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Rendering point clouds with compute shaders and vertex order optimization. *Computer Graphics Forum*, 40(4):115–126, 2021.

[47] Scratchapixel. Overview of the rasterization algorithm. `https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html/`, n.d. Accessed: 2026-01-08.

[48] Chi-Yi Tsai and Chih-Hung Huang. Indoor scene point cloud registration algorithm based on rgb-d camera calibration. *Sensors*, 17(8), 2017.

[49] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing '93*, pages 12–21, 1993.

[50] Eric W. Weisstein. Euler's rotation theorem. `https://mathworld.wolfram.com/EulersRotationTheorem.html`.

[51] Eric W. Weisstein. Rotation. `https://mathworld.wolfram.com/Rotation.html`.

[52] Eric W. Weisstein. Rotation matrix. `https://mathworld.wolfram.com/RotationMatrix.html`.

[53] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.