UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Visualització de Point Clouds: Mètode de Newton en raigs cònics

Autor: Pol Sánchez Forns

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

Barcelona, 17 de gener de 2024

Abstract

This Final Degree Project focuses on the study and implementation of Newton's method in conical rays. To do this, it explores the concept of point clouds, as well as the understanding of voxelization and octrees processes, which serve as a foundation for the efficient representation and manipulation of point clouds. It also explores various visualization methods, including splatting, point representation as spheres, as well as ray-based techniques modeled as cones or cylinders. The theoretical aspects behind the iterative method are highlighted, providing an algorithm capable of applying it. Part of the TFG focuses on the efficient transfer of point cloud data to graphic processing units (GPUs), exploring the architecture of the program and its optimization structures. Finally, the results obtained when applying Newton's method for conical rays (establishing several parameters and using different optimization methods) are shown to compare it with the point-sphere method.

Resum

Aquest Treball de Final de Grau es centra en l'estudi i la implementació del mètode de Newton en raigs cònics. Per a això s'indaga en el concepte de point clouds, així com en la comprensió dels processos de voxelització i octrees, que serveixen com a fonament per a la representació i manipulació eficient dels point clouds. També explora diversos mètodes de visualització, incloent-hi splatting, representació dels punts com esferes, així com tècniques basades en raigs modelats com a cons o cilindres. Es destaquen els aspectes teòrics darrere el mètode iteratiu, proporcionant un algorisme capaç d'aplicar-lo. Una part del TFG es centra en la transferència eficient de dades de point clouds a unitats de processament gràfic (GPU), explorant l'arquitectura del programa i les seves estructures d'optimització. Finalment, es mostren els resultats obtinguts a l'hora d'aplicar el mètode de Newton per a raigs cònics (establint diversos paràmetres i usant diferents mètodes d'optimització) per a comparant-lo amb el mètode de punts esferes.

Resumen

Este Trabajo Fin de Grado se centra en el estudio e implementación del método de Newton en rayos cónicos. Para ello, explora el concepto de point clouds, así como la comprensión de los procesos de voxelización y octrees, que sirven de base para la representación y manipulación eficiente de point clouds. También explora varios métodos de visualización, incluyendo el splatting, la representación de puntos como esferas, así como técnicas basadas en rayos modelados como conos o cilindros. Se destacan los aspectos teóricos que subyacen al método iterativo, proporcionando un algoritmo capaz de aplicarlo. Parte del TFG se centra en la transferencia eficiente de datos de point clouds a unidades de procesamiento gráfico (GPU), explorando la arquitectura del programa y sus estructuras de optimización. Finalmente, se muestran los resultados obtenidos al aplicar el método

de Newton para rayos cónicos (estableciendo varios parámetros y utilizando diferentes métodos de optimización) para compararlo con el método de los puntos esféricos.

Agraïments

Vull agrair a la meva família, que sempre m'ha fet costat i estimat; als meus amics, que m'han soportat; a l'Anna, que ha tingut paciència infinita amb mi; i en Gerard Perelló, que m'ha ajudat per a qualsevol dubte que tenia.

Índex

Introducció

Context i motivació

La computació gràfica ha patit un creixement substancial en els últims anys, principalment atribuïble als avenços en les GPUs. Aquest progrés ha permès visualitzar objectes i escenes 3D en pantalles, fent de la computació gràfica una eina essencial per a diversos propòsits, que van des del desenvolupament de videojocs fins a sensors per a cotxes autònoms.

Un mètode prevalent per modelar objectes 3D és a través de *point clouds*, que són conjunts de punts en l'espai 3D distribuïts no-uniformement. El seu objectiu és representar qualsevol superfície geomètrica d'objectes 3D amb el mínim d'espai possible.

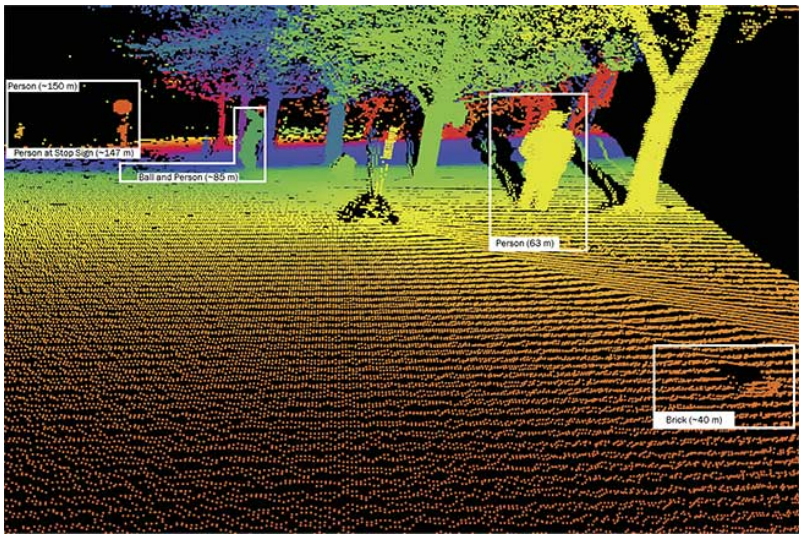

Figura 1: Visualització del que detecta el LIDAR d'un cotxe autònom, i la seva Segmentació semàntica. Foto extreta del blog d'en Greg Smolka [14]

Els *point clouds* es generen sovint utilitzant tecnologies com els escàners LIDAR o la fotogrametria. Els escàners **LIDAR** llencen raigs a la superfície que vol escanejar, i, segons el que tardi el raig en retornar a l'escàner, i el seu angle de sortida, es representa el punt

en un lloc determinat de l'espai. La fotogrametria, per l'altra banda, consisteix en fer diverses fotos d'un mateix objecte des de diferents punts de vista. Llavors s'agafen punts coincidents en les tres fotos i 'triangulen' la posició del punt en l'espai tridimensional. Aquesta última tècnica s'usa juntament amb el mètode de visualització de splatting (el qual es detalla en aquest treball) per a generar escenes realistes tridimensionals, com es mostra en [6]. Ambdós mètodes ofereixen una major precisió en la captura i recreació d'entorns 3D, fent-los adequats per a tasques com la segmentació 3D.

Altres aplicacions en la visualització de point clouds són els escàners dels cotxes autònoms, com s'explica en aquest blog [3]. Aquests s'usen perquè els cotxes puguin identificar i localitzar objectes, per poder prendre decisions ben informades sobre la seva trajectòria i velocitat. A més, els point clouds contribueixen significativament a la **Segmentació Semàntica**, on cada punt del point cloud es classifica en funció de la seva etiqueta semàntica (per exemple, carretera, construcció, arbre). Aquesta classificació permet als vehicles autònoms comprendre els matisos estructurals de l'escena i distingir entre diferents tipus d'objectes, facilitant en última instància processos de presa de decisions més robustos.

Objectiu principal

L'objectiu principal d'aquest treball és estudiar i aplicar un mètode iteratiu de visualització de point clouds, per a comparar-lo amb el mètode usat en el treball del Gerard Perelló [13]. A més, es vol veure quin mètode d'optimització va millor per a aquest mètode.

Objectius específics

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

- Entendre en profunditat què són els point clouds i quina estructura tenen.
- Estudiar els diferents mètodes d'optimització de point clouds que existeixen.
- Explorar els diferents mètodes existents per a la visualització de point clouds.
- Aprofundir en el mètode iteratiu de Newton pel raig cònic, estudiant-lo i codificant-lo.
- Veure quin és el seu rendiment depenent de les petites variacions que pot tenir, i comparant-lo amb un mètode alternatiu.

Planificació temporal

Com es mostra a la figura 2, el procés s'ha planificat durant aproximadament 7 mesos. L'etapa més llarga ha estat la de recerca. Però sobretot la feina s'ha concentrat durant l'últim mes abans de l'entrega, per l'impossibilitat de fer-ho durant el semestre. Gràcies a la

feina feta pel Gerard Perelló en el seu treball [13], ha estat molt més senzill la comprensió i la codificació del mètode que s'ha estudiat.

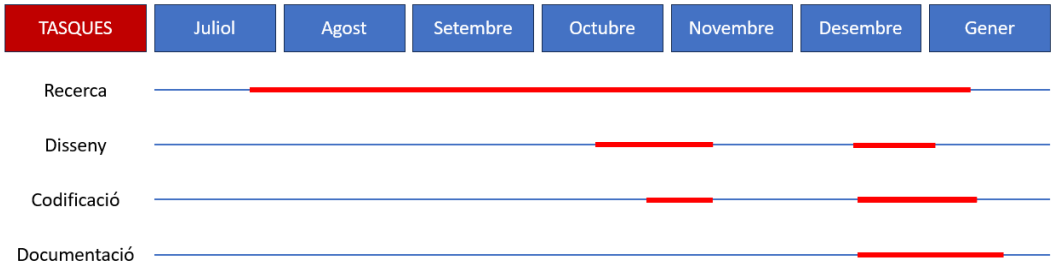

Figura 2: El diagrama de Gantt del treball planificat de Juliol a Gener

Organització de la memòria

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

- **Introducció:** En aquest capítol s'explora el context d'aquest projecte i s'estableix els objectius del treball.
- **Antecedents**: Aquí s'explica el problema que es vol resoldre, s'estudien les possibles solucions d'aquests.
- **Disseny:** En el capítol de Disseny es descriu el disseny gràfic i l'arquitectura de software usat.
- **Resultats i Simulacions:** En aquest el capítol es mostren els resultats del projecte desenvolupat.
- **Conclusions i feina futura:** En aquest últim capítol es detallen les conclusions i l'estat del projecte de cara al futur.
- **Apèndix: Manual Tècnic:** En el Manual Tècnic es descriuen tots els passos detallats de com instal·lar l'aplicatiu.

Capítol 1

Antecedents

Abans d'aprofundir en detalls, necessitem veure quins són els desafiaments que ens trobem a l'hora de visualitzar un point cloud, i solucions per a aquests problemes. També s'explorarà alguns mètodes que existeixen per a renderitzar point clouds.

1.1 Definició del problema

Des de fa més de tres dècades que s'estudien els point clouds. El 1985, Levoy i Whitted [11] van proposar la idea d'utilitzar els punts com una primitiva geomètrica per a la representació, com una alternativa a les malles poligonals o superfícies paramètriques, ja que a mesura que el nombre de figures geomètriques en les escenes (i.e. primitives) creixia, les seves mides projectades podrien veure's reduïdes en àrees tan petites, que a l'hora de projectar-les a la pantalla ocuparien menys espai que un píxel. Així, representar les superfícies amb un conjunt de punts podria donar flexibilitat a l'hora de projectar la superfície usant més o menys punts segons la seva àrea de projecció. A més, els point clouds són més eficients a l'hora de visualitzar-se que les malles, i es poden emmagatzemar i visualitzar grans conjunts de dades amb fins a 100 milions de punts en temps real. No obstant això, també hi ha problemes.

En el cas que usem raytracing, trobar el punt d'intersecció entre un raig i la representació implícita de la superfície del point cloud no és un problema trivial. Això es deu al fet que la probabilitat que un raig xoqui contra un punt del point cloud qualsevol és 0. Així que no podem aplicar directament l'algorisme de raytracing. Hi ha altres estratègies per poder solucionar aquest problema. Una manera és reconstruint tot el point cloud en una malla triangulada, i després aplicar el raytracing de forma convencional. Per a point clouds amb pocs punts serveix, però per a point clouds de l'ordre de desenes de milions de punts és poc eficient. A més, en certs casos més complexes aquest procés pot ser més complicat i pot provocar la pèrdua d'informació. Una altra idea és canviar la dimensionalitat dels punts o del raig perquè la probabilitat d'intersecció no sigui nul·la. Els exemples que s'explicaran més endavant és canviant els punts per quadrats bidimensionals enfocats

a la càmera, o canviant-los per esferes. I també hi ha estratègies que canvien el raig per un cilindre o un con. Totes aquestes estratègies es discutiren en aquest capítol.

Un altre dels problemes és la gran quantitat de punts que pot arribar a tenir un point cloud. Comprovar per a cada un dels punts si creua el raig, o rasteritzar cada un dels punts pot ser molt costós. Per tant, hem de crear estructures que redueixi el conjunt de punts candidats a ser projectats en un píxel. En aquest treball en parlarem de dues estructures, la voxelització i l'octree.
A continuació s'introdueixen els conceptes bàsics i les principals tècniques de representació i compactació de point clouds per a tot seguit explicar les diferents estratègies de visualització de point clouds.

1.2 Definició de conceptes

Abans d'entrar en detalls, s'han de definir en detall cada un dels nous conceptes que surten en aquest treball. Comencem veient a què ens referim quan parlem de point clouds.

Definition 1.1. *Un **point cloud** és un conjunt de punts* $\{p_1, p_2, ..., p_n\}$ *definits en* $\mathbb{R}^3$.

Com ja s'ha mencionat, els point clouds són representacions de superfícies tridimensionals que contenen un nombre finit de punts, generalment en l'ordre dels milions. Aquestes col·leccions de punts no es troben distribuïdes uniformement sobre la superfície implícita del point cloud. A més, cada punt té atributs associats. En aquest treball, cada punt p del point cloud es caracteritza pels següents atributs:

1. **Normal:** El vector normal de la superfície implícita en aquell punt. Es nota com $\vec{n_p}$.
2. **Color:** El color associat al punt. Es nota com c_p.
3. **Etiqueta:** Donat un conjunt de n etiquetes $\mathcal{L} = \{eti_1, eti_2, ..., eti_n\}$, podem fixar una d'aquestes etiquetes al punt.

A vegades va bé pensar aquest concepte com una estructura de dades, recopilant tota la informació que ens interessa del punt:

Definition 1.2. *Definim com a **point** $\bar{p} := \{p, \vec{n_p}, c_p, l_p\}$ l'estructura que representa a cada punt del point cloud.*

1. *p: Són les coordenades del punt p. Es representa com una tupla de 3 coordenades (x, y, z).*
2. *$\vec{n_p}$: Són les coordenades del vector $\vec{n_p}$. Es representa com una tupla de 3 coordenades (n_x, n_y, n_z).*
3. *c_p: És una tupla de 3 coordenades (R, G, B). Les tres primeres coordenades són valors del 0 a l'1 representant quan intervé a la barreja de colors el vermell, el verd i el blau, respectivament.*
4. *l_p: Un vector de dimensió igual al nombre d'etiquetes disponibles. Si l'etiqueta associada al punt és l'eti_i, llavors aquest vector tindrà totes les coordenades igual a 0 excepte la coordenada i. Es representa com una tupla de 3 coordenades (x, y, z).*

1.3 Optimització dels point clouds

A causa de la gran quantitat de punts que pot arribar a haver-hi en un point cloud, és necessari usar eines per poder reduir el temps de renderització al màxim. En aquest treball s'usen dues tècniques d'agrupació de point clouds per poder dividir el problema en petits trossos a visualitzar.

La primera consisteix en els **vòxels**, els quals es representen com petites capses, simulant un píxel en 3 dimensions. Es dividix la capsa contenidora del point cloud en vòxels i es suposa que allà els punts estan uniformement distribuïts.

La segona consisteix a crear capses contenidores més petites per a grups més petits de punts. Llavors es forma una jerarquia de capses contenidores fins a arribar a una capsa contenidora mínima, la qual contindrà un petit conjunt de punts. Aquesta estructura s'anomena **octree**.

En aquesta secció s'explicarà com es construeixen aquestes optimitzacions i com s'implementen per a un point cloud qualsevol.

1.3.1 Voxelització

Voxelitzar un point cloud significa dividir la caixa contenidora del point cloud en cubs de costat b, i tot seguit, agrupar tots els punts que estiguin dins del mateix vòxel. És l'equivalent tridimensional a la pixelització per a imatges 2D. De fet, la paraula 'vòxel' prové de l'acrònim de 'volum' i 'píxel'. Això es deu al fet que volem que el vòxel sigui la unitat mínima d'informació de l'objecte. La figura 1.1 mostra un exemple d'una voxelització d'un point cloud representant un conill.

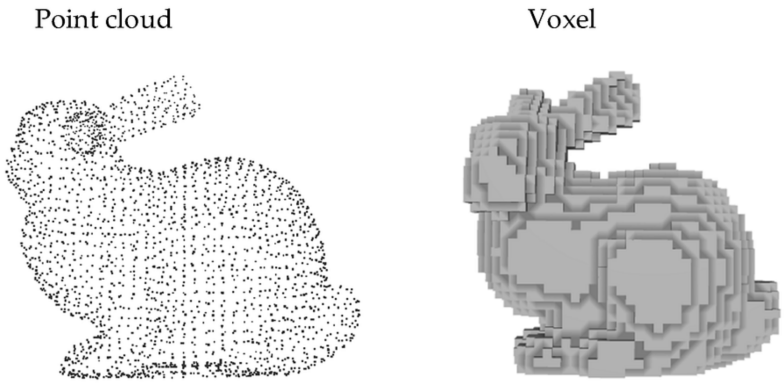

Figura 1.1: A l'esquerra es mostra un point cloud que representa un conill. A la dreta, la seva voxelització.

Per fer més eficaç l'explicació en aquesta subsecció, farem servir notació nova. Siguin $v = (v_x, v_y, v_z)$, $u = (u_x, u_y, u_z)$ dos vectors de l'espai. Direm que $u \leq v$ sii $u_x \leq v_x$, $u_y \leq v_y$ i $u_z \leq v_z$. A més, donada una funció $f : \mathbb{R} \to \mathbb{R}$, es farà abús de notació

considerant $f(v) = (f(v_x), f(v_y), f(v_z))$. Per exemple, l'usarem per a la funció sòl $\lfloor \cdot \rfloor$ més endavant (per poder definir el vòxel al que pertany un punt del point cloud).

Un algorisme de voxelització és el següent. Suposem que la capsa mínima contenidora està delimitada pels vèrtexs v_{min} i v_{max}. Llavors, els vòxels, si volem que tingui costat b, estaran limitats pels vèrtexs:

$$v_{min}^{i,j,k} = b[i,j,k] + v_{min},\ v_{max}^{i,j,k} = b[i+1,j+1,k+1] + v_{min}$$

on i, j, k són enters. Amb aquesta notació, podem identificar cada vòxel amb una coordenada $[i,j,k] \in \mathbb{Z}^3$. Llavors el vòxel $[i,j,k]$ és aquell que està i vòxels en sentit x, j vòxels en sentit y i k vòxels en sentit z, com es mostra en la figura 1.2.

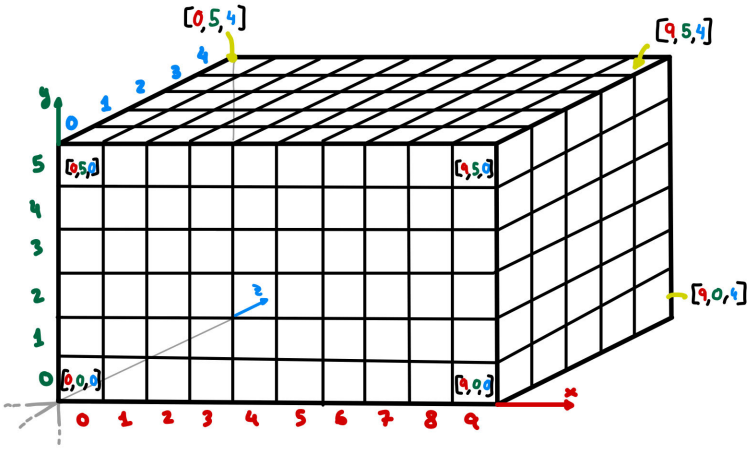

Figura 1.2: Mostra de les coordenades dels vòxels

Aquests valors varien de 0 a s_x, s_y, i s_z pel nombre de vòxels en cada dimensió. Notem s com el vector (s_x, s_y, s_z). En l'exemple de la figura 1.2, $s = (10, 6, 5)$.

Ara calculem s i determinem a quin vòxel pertany cada punt del point cloud. Per aconseguir-ho, necessitem expressar matemàticament la condició que un punt estigui contingut en una caixa. Si considerem u_{min}, u_{max} com a vèrtexs delimitadors, llavors el punt p es troba dins la caixa si $u_{min} \leq p \leq u_{max}$.

Primer trobem la forma de veure quin és el vòxel que pertany un punt p. Tenim que, pel que hem vist abans, el punt p pertany al vòxel $[i,j,k]$ si:

$$b[i,j,k] + v_{min} \leq p \leq b[i+1,j+1,k+1] + v_{min} \Rightarrow [i,j,k] \leq \frac{1}{b}(p - v_{min}) \leq [i+1,j+1,k+1]$$

fixem-nos que, com i, j, k són enters, això és equivalent a dir que:

$$[i,j,k] = \lfloor \frac{1}{b}(p - v_{min}) \rfloor$$

El que ens permet trobar el vòxel el qual p pertany.

Cal fixar-nos en el fet que la voxelització no té per què cobrir exactament la capsa contenidora, sinó que pot sobresortir. En qualsevol cas, volem que el vòxel que cobreixi el vèrtex v_{max} de la capsa contenidora sigui el 'darrer' en les tres dimensions, és a dir, el vòxel $[s_x - 1, s_y - 1, s_z - 1]$. Per tant:

$$b[s_x - 1, s_y - 1, s_z - 1] + v_{min} \leq v_{max} \leq b[s_x, s_y, s_z] + v_{min}$$

Reordenant obtenim que

$$[s_x - 1, s_y - 1, s_z - 1] \leq \frac{1}{b}(v_{max} - v_{min}) \leq [s_x, s_y, s_z]$$

Que, equivalentment al que hem fet abans, tenim que:

$$\lceil \frac{1}{b}(v_{max} - v_{min}) \rceil = [s_x, s_y, s_z]$$

Trobant així una fórmula per calcular el vector s.

1.3.2 Octree

En informàtica i geometria computacional, un octree és una estructura de dades basada en arbres octals utilitzada per a la partició de l'espai tridimensional en regions més petites, com es mostra en la figura 1.3. Representa i organitza de manera eficient dades volumètriques de manera jeràrquica.

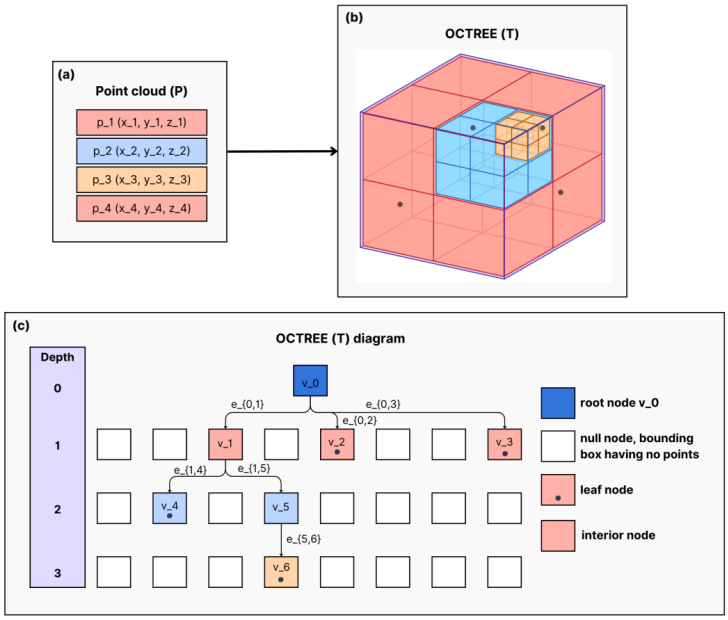

Figura 1.3: Esquema sobre la creació de l'octree a través. Figura extreta del treball del Gerard Perelló [13]

El node base és la capsa contenidora, i els possibles nodes fills de cada node són les 8 subcapses amb la meitat de longitud de costat que cobreixen el node pare.

Cada node té els següents atributs:

- **points:** Vector de *points*. Són els punts del point cloud dins del node.
- **minBound, maxBound:** Són els vèrtexs que determinen la capsa del node.
- **leaf:** Booleà que determina si el node és un node fulla o no
- **children:** Vector de 8 nodes. Aquí es guarden els fills del pròpi node. No tot node té 8 fills, així que no sempre està plena.

No són els únics atributs que té, però són els que usarem per veure com es crea l'octree.

Per generar un octree, primer inicialitzem el node base, amb tots els punts del point cloud i indicant els vèrtex de la capsa contenidora. Tot seguit apliquem la funció `build`, especificada en l'algorisme 1, al node en qüestió.

Algorithm 1 Creació de l'Octree

Input: Node de l'Octree: *node*.

```
1:  function BUILT(OctreeNode* node)
2:      if node is leaf then
3:          node->leaf = True;
4:          return;
5:      for each subBox of node do
6:          vector<Points> childPoints;
7:          for point in node->points do
8:              if point in subBox then
9:                  Add point to childPoints;
10:         if childPoints has points then
11:             OctreeNode childNode(subBox, childPoints)
12:             Add childNode to node->children;
13:             build(childNode);
```

La funció segueix el típic algorisme de construcció d'arbres. Primer de tot, en la línia 2, comprova si el node és un node fulla. Per fer-ho es comprova si es compleix una de les següents tres condicions:

- Si la profunditat del node és menor que una profunditat màxima $depth_{max}$ establerta.
- Si el node té un nombre mínim de punts $Npoints_{min}$ especificat.
- Si tots els punts del node tenen la mateixa etiqueta.

Si es veu que no és un node fulla, mirem cada una de les subcapses del node (línia **5**), i trobem els punts que estan dins d'aquesta (línies **6-9**). Si hi ha com a mínim un punt (es

comprova en la línia **10**), creem el node fill, inicialitzat amb la subcapsa contenidora i els punts recol·lectats en el for de la línia **6**. A partir d'aquí apliquem la funció `build` de nou (línia **13**) per construir així al complet l'octree.

En comparació amb la voxelització, veiem que emmagatzem molta més informació i que, a més, l'estructura és molt més complexa. Però és molt més senzill d'usar, per la seva jerarquia. Aquesta, a més, minimitza les comprovacions gràcies a la variabilitat de mida dels nodes fulla.

1.4 Estratègies de visualització de point clouds

Com ja s'ha vist abans, els point clouds no es poden visualitzar usant el mateix principi que al renderitzar malles o superfícies implícites, a causa de la dimensionalitat dels punts i els raigs. Per tant, volem veure quines estratègies tenim per renderitzar-los, i discutir quin és el que usarem en aquest treball.

1.4.1 Splatting

El **splatting gaussià** [16] és una tècnica de renderització de point clouds molt semblant al Neural Radiance Fields (NeRF). Destaca sobretot per la seva impressionant velocitat de renderització, i la fidelitat que pot arribar a aconseguir en la visualització, usant aprenentatge automàtic.

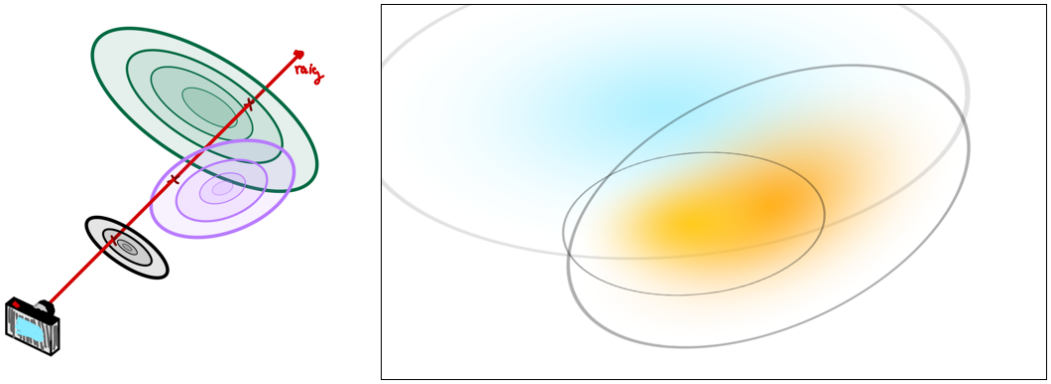

Figura 1.4: Exemple de splatting. La foto emmarcada ha estat extreta de l'article de Dylan Ebert [8]

La idea principal de la tècnica consisteix en representar els punts com a funcions gaussianes tridimensionals. Cada un d'aquests punts tenen definits els seus propis paràmetres, que especifiquen la gaussiana que els representa. Tenim la **mitjana** μ, la **matriu de covariància** Σ i l'**opacitat** $\sigma(\alpha)$, una funció sigmoide amb imatge en l'interval $[0, 1]$.

$$f_i(p) = \sigma(\alpha_i) \exp(-\frac{1}{2}(p - \mu_i)\Sigma_i^{-1}(p - \mu_i))$$

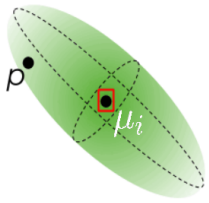

Figura 1.5: Funció gaussiana que representa el punt σ_i

Aquests paràmetres tenen un impacte significatiu en com es visualitza el punt que representa. En la figura 1.5, veiem com cada un dels paràmetres descrits juguen per donar-li forma, orientació i posició a l'el·lipsoide. El punt emmarcat és la mitjana, i correspon al punt representatiu. La matriu de covariància és la que determina les el·lipses discontínues, que són les que estableixen la forma i l'orientació de l'el·lipsoide que forma la funció. Finalment, la funció σ determina la transparència de l'objecte.

En general, s'usen en point clouds extrets a partir de fotogrametria. És a dir, a través de diverses imatges es triangula la posició de punts. Per tant, tenim diversos punts de referència des d'on sabem com s'ha de veure el renderitzat. Usant això de base, es pot entrenar una xarxa neuronal o usar el mètode de Descens del Gradient Estocàstic per a establir els paràmetres de cada punt perquè minimitzin l'error. Recentment, s'han fet molts avenços en la investigació d'aquesta tècnica, com es mostra en la web [6].

1.4.2 Punts esferes

Una altra tècnica més intuïtiva és considerar els punts com esferes. Cada un d'aquests punts tindrà associat un radi, i una transparència, com en el cas de l'splatting. A partir d'aquí, es faria raytracing. Com tenen una component de transparència, el raig seguirà fins que la suma de transparències superi a 1.

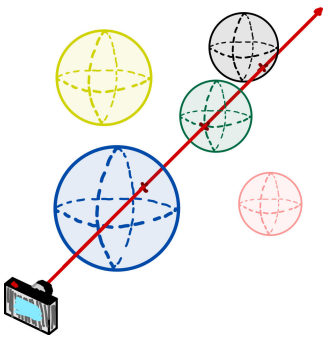

Figura 1.6: Exemple visualització amb punts esferes

A la figura 1.6 es mostra un exemple. En aquest cas, el raig veuria el color ponderat de les tres esferes, on el seu pes depèn de la transparència.

Tot i que aquesta estratègia no busqui representar amb la màxima fidelitat la realitat, els càlculs són més senzills que els requerim per al splatting. A més, afavoreix a la distinció de punts, fent més fàcil la visualització d'algorismes de segmentació, per exemple. Aquesta és l'estratègia escollida pel Gerard Perelló en el seu treball [13].

1.4.3 Raigs cilíndrics o cònics

La idea d'aquest mètode és veure quins són els punts que afectarien en el color que es veu en un píxel. Així que volem trobar el subespai de l'espai que veu la càmera que acabarà en el mateix píxel. Aquesta zona es pot aproximar com un cilindre infinit, amb eix el raig, i un radi determinat. Una altra forma es aproximant-lo com un con infinit, amb l'eix el raig i amb un angle determinat. En la figura 1.7 es veuen ambdues representacions.

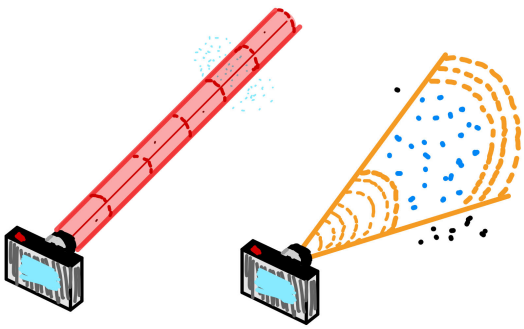

Figura 1.7: Representació del raig com un cilindre (esquerra) i com un con (dreta)

Llavors, es pot fer el càlcul del color del píxel tenint en compte els atributs dels punts dins l'espai. Un exemple es fer un color ponderat, on els pesos depenen de la distància del punt al raig, o a la càmera.

1.4.4 Conclusions

En aquesta secció, s'han explorat diverses tècniques de renderització per a la visualització de punts en entorns tridimensionals. El **splatting gaussià** destaca per la seva eficiència i fidelitat en la renderització de núvols de punts, utilitzant funcions gaussianes tridimensionals per representar cada punt. Aquest enfocament permet una representació detallada de la realitat més fidedigna.

D'altra banda, s'ha introduït la tècnica de representar **punts com a esferes**, amb els quals es realitza raytracing. Aquesta metodologia, tot i no buscar la màxima fidelitat a la realitat, ofereix càlculs més simples i facilita la visualització d'algorismes de segmentació gràcies a la distinció clara entre punts.

Finalment, s'ha presentat la idea d'utilitzar **raigs cilíndrics o cònics** per aproximar la zona que afecta el color d'un píxel. Aquest mètode interpreta el subespai visual des de la càmera del píxel com un cilindre o con infinit amb atributs específics.

Considerant aquesta exploració, es conclou que la representació mitjançant **raigs cònics** es planteja com la més idònia per aprofundir en el desenvolupament de treball. Aquesta elecció es fonamenta en el fet que la base es la més intuïtiva a l'hora de pensar sobre com veure els point clouds amb un gran nombre de punts. Aquest mètode el compararem amb el mètode dels **punts esferes** en termes d'eficiència i de resultats visuals.

Capítol 2

Mètode de Newton en raigs cònics

En aquest capítol en detall quin és l'algorisme que usem per veure point clouds. Està inspirat en els algorismes presentats en els treballs [10], [9], [5] i [4]. En el nostre cas, representem el nostre raig com un con, així que els punts afectaran el píxel associat al raig seran aquells que estiguin dins del con. A més, gràcies a la modificació de certes variables de l'algorisme, podrem modificar la imatge final. S'explicarà cada una de les variables i en què afecten la visualització.

2.1 Justificació de l'algorisme

Com ja s'ha comentat en la secció anterior, el point cloud té associada una superfície implícita. Llavors la idea de l'algorisme que presentarem és generar una aproximació de la superfície únicament tenint en compte els punts dins del raig çon", creant aproximacions polinòmiques **locals** a través de l'espai per definir aquesta superfície.

Així que sigui el point cloud el conjunt $\{p_i \in \mathbb{R}^3 \mid i \in \mathbb{N}_{\leq n}\}$ i n_i la normal en el punt p_i, tal com s'ha definit a la secció 1.1. Agafem un subconjunt I de $\mathbb{N}_{\leq n}$, fent que $P_I = \{p_i | i \in I\}$ sigui un subconjunt del point cloud. A més, suposem que aquesta superfície S és infinitament diferenciable.

L'eina clau per a aquesta definició és la mitjana ponderada de punts dins d'un entorn de la superfície. Aquesta dependrà d'una funció de pes $\theta : \mathbb{R}^+ \to \mathbb{R}^+$, que especificarà la influència que té un punt p_i sobre la mitjana, fent servir la distància euclidiana d'aquest punt a un punt x. És a dir, tindrem que $\theta_i(x) := \theta(\|p_i - x\|)$ és el pes que tindrà el punt p_i en la mitjana. I com que en el nostre cas volem que com més a prop estigui un punt de x, més influència tingui, la funció de pes ha de ser monòtona decreixent (com a mínim en el domini dels reals positius) i infinitament diferenciable. La mitjana ponderada dels punts respecte al punt x, la qual anomenarem el **baricentre** respecte x, s'obté amb la següent fórmula:

$$a(x) = \frac{\sum_{i \in I} \theta_i(x) \cdot p_i}{\sum_{i=1}^{n} \theta_i(x)}$$

També voldrem calcular la mitjana ponderada de les normals en els punts, fent que tinguin el mateix pes que els seus punts associats. Volem que aquest vector, el qual anomenarem **baricentre normal** respecte x, estigui normalitzat, així que el calcularem de la següent forma:

$$n(x) = \frac{\sum_{i \in I} \theta_i(x) \cdot n_i}{\|\sum_{i \in I} \theta_i(x) \cdot n_i\|}$$

Llavors es definirà la superfície implícitament utilitzant els dos baricentres mostrats. Aquesta funció implícita π calcula la distància d'un punt x al baricentre respecte x al llarg de la direcció del baricentre normal:

$$\pi(x) = \langle n(x), a(x) - x \rangle$$

I tenim que la superfície és el conjunt de zeros d'aquesta funció implícita:

$$S = \{x \mid \pi(x) = 0\}$$

Per tant, volem trobar, si existeix, el punt al llarg del raig $r(t) := r_0 + t \cdot \vec{r}$ on s'anul·li en π. Una manera eficient de trobar aquest punt és fer servir el **mètode de Newton** per a la funció $f := \pi \circ r$. Això significa que, donat un valor inicial t_0, apliquem la següent fórmula per trobar valors successius de $t_1, t_2, \ldots$ fins a arribar a un punt de convergència:

$$t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)}$$

El problema rau a calcular la derivada de f, ja que calcular la derivada de la funció π és realment complex. Per tant, hem de fer suposicions addicionals per simplificar el càlcul d'aquesta derivada.

Fixem-nos en el point cloud localment, és a dir, en un entorn suficientment petit dels punts que cauen dins del raig con. En aquest entorn, podem considerar els baricentres respecte x quasi constants. En tal cas tenim que la derivada de $f = \langle n(r(t)), a(r(t)) - r(t) \rangle = \langle n(r(t)), a(r(t)) \rangle - \langle n(r(t)), r(t) \rangle$ és $-\langle n(r(t)), \vec{r} \rangle$. Per tant, si notem $x_i = r(t_i)$, tenim que la fórmula de convergència es transforma en:

$$t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)} \approx t_i + \frac{\langle n(x_i), a(x_i) - x_i \rangle}{\langle n(x_i), \vec{r} \rangle} = \frac{\langle n(x_i), t_i \cdot \vec{r} + a(x_i) - x_i \rangle}{\langle n(x_i), \vec{r} \rangle} = \frac{\langle n(x_i), a(x_i) - r_0 \rangle}{\langle n(x_i), \vec{r} \rangle}$$

Si ens fixem, el que estem fent és trobar per a quin t_{i+1} tenim que x_{i+1} és la intersecció entre el raig i el pla $\bar{\pi}_i := \{y \in \mathbb{R}^3 \mid \langle n(x_i), a(x_i) - y \rangle\}$.

Per tant, el nostre objectiu és, donat un x_0 suficientment a prop de la superfície implícita, anar trobant la seqüència de x_i's definits com:

$$x_{i+1} = \bar{\pi}_i \cap r$$

on $r = \{r(t) \mid t \in \mathbb{R}\}$, i identifiquem a x_{i+1} com a l'únic element de la intersecció, en cas d'existir. Aquest procés iteratiu convergirà quan trobem un x_i suficientment a prop de la superfície implícita, és a dir, quan $|\pi(x_i)| < \epsilon$, per a un ϵ suficientment petit. En la següent secció veurem quan aquest algorisme divergeix.

La figura 2.1 es veu una iteració d'exemple de l'algorisme. En la primera part veiem la segmentació del point cloud, separant els punts dins del con (blaus) i fora del con (negre). Tot seguit calculem les imatges de les ponderacions dels punts per al baricentre. Un cop calculat, com veiem en la tercera part, mirem la intersecció amb el pla, i tornem a iterar.

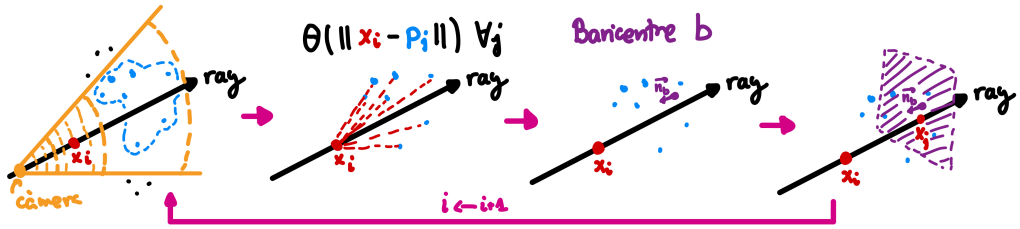

Figura 2.1: Exemple d'una iteració de l'algorisme.

2.2 Descripció de l'algorisme

Tal com hem presentant en la secció anterior, el mètode que volem implementar és un iteratiu sobre punts en el raig. Si aquest convergeix, tindrem hit. I si divergeix, aquell raig no xocarà amb el point cloud. Aquest és l'algorisme 2.

Abans d'explicar en detall l'algorisme, s'ha de veure el significat de la notació presa. Primer de tot, definim el P' (el conjunt de punts que contribueixen en el càlcul dels baricentres) com el subconjunt $Q \cap C(ray)$, on

$$C(ray) := \{y \in \mathbb{R}^3 \mid \langle y - r_0, \vec{r} \rangle > \|y - r_0\| \cdot \|\vec{r}\| \cdot \gamma\}$$

és el con infinit de direcció $\vec{r}$, vèrtex r_0 i tal que el cosinus de l'angle del con és igual a γ.

Després tenim la funció $WA(x, P', \theta)$ retorna el *point* ponderat. Aquest tindrà el baricentre de cada una dels elements. Les coordenades d'aquest point serà les coordenades de $a(x)$, i la normal és el vector $n(x)$, ambdós presentats en la primera secció. Pel que fa al color, una mitjana ponderada de les seves coordenades, amb els pesos definits anteriorment, compleix els límits establerts per definició. Ara bé, en operar amb el vector d'etiqueta del *point*, hi ha una problemàtica, ja que el resultat no és una etiqueta. El resultat és un vector amb valors entre 0 i 1, on la suma de les coordenades és sempre 1. Això

Algorithm 2 Mètode iteratiu del raig

Input: Raig: *ray*, t mínima: t_{min}, t màxima: t_{max}, HitInfo per al RT: *hit*, subconjunt del point cloud: Q, Funció de pes: θ.
Output: Booleà que determina si el raig xoca amb el point cloud.

```
1:  if P' == ∅ then
2:      return false;
3:  float t ← t_min;
4:  vec3 x ← r(t);
5:  while número d'iteracions < 100 do
6:      Point p̄ ← WA(x, P', θ)
7:      π̄(x) := ⟨p̄.position − x, p̄.normal⟩
8:      if |π̄(x)| < ε then
9:          hit ← x, t, p̄;
10:         return true;
11:     else
12:         t ← t tal que r(t) ∈ π̄
13:         if t ∉ [t_min, t_max] then
14:             return false;
15:         x ← r(t);
```

permet interpretar la coordenada i del vector resultant com la probabilitat que el *point* resultant tingui l'etiqueta eti_i. Per tant, considerarem com a etiqueta del *point* el vector on la coordenada amb valor 1 és la coordenada de major probabilitat del resultat. Com veiem, aquest *point* ponderat depèn de la funció de pes θ, presentada en la secció anterior.

En la línia 6 es defineix la funció $\bar{\pi}$. El conjunt de punts que s'anul·len en $\bar{\pi}$ és el pla amb punt base $a(x)$ i normal $n(x)$, el qual notarem igual que la funció. Tal com hem vist en la secció anterior, volem trobar la intersecció entre aquest pla i el raig per trobar el següent punt de la iteració. Cal recalcar que es compleix que $\bar{\pi}(x) = \pi(x)$.

Veiem com fa l'algorisme per aplicar el mètode presentat abans. Primer de tot, si es té que el raig con no interseca amb el subconjunt desitjat Q, retornarà que no hi ha hagut *hit* (línies **1,2**). Un cop l'algorisme s'assegura que el conjunt P' no és buit, s'estableix x com el punt del raig de paràmetre t_{min}. Aquest valor, igual que t_{max}, dependrà de l'optimització que apliquem en l'algorisme, però en qualsevol cas ha de fer que x estigui suficientment a prop del point cloud.
Tot seguit s'entra en un bucle, el qual representa les iteracions del mètode de Newton. S'estableix un màxim de 100 iteracions per a la convergència (línia **4**). En cas que necessités més iteracions, es considerarà que el mètode divergeix. Llavors, per a cada iteració del bucle, primer s'estableix el *point* baricentre del subconjunt P' respecte al punt x. Es fa la comprovació per veure si ja ha convergit (i.e., si $|\bar{\pi}(x)| = |\pi(x)| < \epsilon$). En cas de no haver-ho fet, s'actualitzarà el punt x del raig com la intersecció del raig amb el pla $\bar{\pi}$. Per fer-ho, primer es calcula el nou paràmetre t. Si aquest està fora de l'interval $[t_{min}, t_{max}]$,

considerarem que el mètode ha divergit i es retorna que el raig no ha xocat contra el point cloud. En cas contrari, es continua amb les iteracions. Si en alguna de les iteracions el mètode convergeix i s'estableix el punt x com el punt de xoc entre el raig i el point cloud. La normal i el color en aquell punt seran la normal i el color del baricentre. S'actualitzarà la variable hit perquè contingui tota la informació de la intersecció (el punt d'intersecció, la normal en aquell punt, el color, ...) i es retornarà que el raig ha xocat contra el point cloud.

2.2.1 Variables de l'algorisme

Aquest algorisme presentat té variacions que afecten en la visualització final del point cloud. En podem destacar 4 d'elles:

- **Funció** θ: Com s'ha esmentat prèviament, aquesta funció ha de ser monòtona decreixent (en valors positius) i ser infinitament derivable. Tenim dues opcions per a la funció θ que podem considerar.

  La primera és $\theta(x) = \frac{1}{x+a}$, la qual és fàcil de calcular. El paràmetre a determina el pes donat als valors propers a 0 respecte als altres. Com més gran sigui, menor pes tindran aquests valors. A més, a no pot ser negatiu, ja que això no només crearia una indeterminació en valors positius, sinó que també faria que ja no fos decreixent.

  El segon candidat és la funció gaussiana, $\theta(x) = e^{-\frac{(x-\mu)^2}{\sigma^2}}$, més complexa de calcular, però més flexible a causa dels seus dos paràmetres: μ i σ. Normalment, es fixa $\mu = 0$, que juga un paper semblant al valor a mencionat abans, determinant el pes dels valors propers a 0 comparats amb els altres. Pel que fa al paràmetre σ, controla la 'localitat' de l'aproximació. Valors més petits per a σ produeixen una aproximació més local, mentre que valors més grans poden suavitzar petites variacions en la superfície, com ara el soroll.

- **Tolerància** ϵ. Apareix a la línia 7 del codi i determina la precisió que volem obtenir del nostre punt final. Com en general els point clouds són punts extrets d'una superfície continua, la disminució d'aquest valor fa augmentar el nombre d'iteracions, sense canviar el fet que el raig col·lideixi o no amb el point cloud. Segons el treball [5], si usem la funció gaussiana per a la funció θ, i posem una tolerància de 0.001σ, l'algorisme convergeix en 2.91 iteracions de mitjana. Generalment, s'establirà $\epsilon = 10^{-t}$, per a cert t real positiu.

- **Valor** γ**:** Representa el valor del cosinus de l'angle de la paret del con respecte a l'eix central del con α, com es mostra en la figura 2.2. Aquest valor ha de ser pròxim a 1, i com més a prop estigui més petit seran els punts que veurem en la pantalla. Generalment, s'establirà $\gamma = 1 - 10^{-g}$, per a cert g real positiu.

- **Subconjunt** Q**:** No podem comprovar per a tots els punts del point cloud si estan dins el con o no, ja que això suposaria milions de comprovacions per raig, la qual cosa no és eficient. A més, quan sigui viable, ens agradaria poder agafar un petit

conjunt de punts dins del con representatius de la superfície intrínseca. Aquest subconjunt depèn de l'estratègia d'optimització que usem, com veurem a continuació.

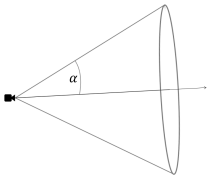

Figura 2.2: Con que representarà un píxel en la càmera

2.3 Optimitzacions amb vòxels

La tècnica de voxelització es pot enfocar de diverses formes, gràcies a totes les possibilitats que ofereix.

Una idea seria tractar als vòxels com capses contenidores més petites. Aprofitar la seva estructura per limitar el nombre total de punts que hem de tenir en compte a l'hora de veure si el raig xoca amb la superfície o no. Però ja apliquem una idea similar aprofitant l'optimització que ofereix l'octree, el qual seria més eficient per l'estructura.

Una altra idea seria considerar els vòxels com la reducció del point cloud. És a dir, pensar els vòxels com a objectes en l'espai, el qual té informació derivada dels punts continguts en aquell vòxel. Per exemple, es podria reduir el nombre de punts que hi ha dins del vòxel i que aquells siguin els representants del vòxel.

Un altre exemple seria considerar el vòxel com el baricentre dels punts de dins. És a dir, calcular les mitjanes ponderades de les coordenades, normals, colors i etiquetes, amb un pes equitatiu per a cada un d'ells, i aplicar l'algorisme només per a aquell punt, si cau dins del con.

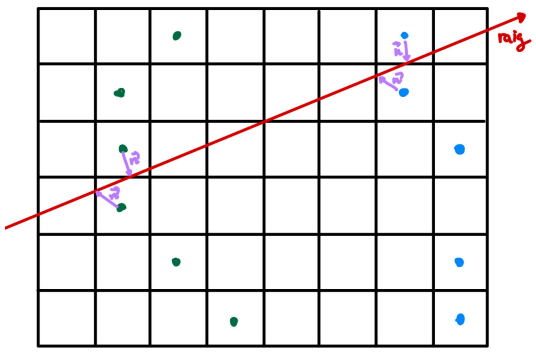

Figura 2.3: Exemple d'aplicació del mètode en una voxelització. Secció 2D

Finalment, però, he decidit considerar el vòxel com un punt en el centre d'aquest. Això simplifica molt els càlculs, i es basa en la premisa que els punts estan més o menys distribuïts uniformement dins del vòxel. La normal la considerarem com el vector que va del centre del vòxel fins a la intersecció entre el vòxel i el raig. En termes generals, és fàcil veure que dona una bona aproximació de la normal de la superfície, si la superfície es densa. El color que representarà serà la mitjana dels colors dels punts de dins del vòxel i l'etiqueta serà l'etiqueta de més 'probabilitat', tal com hem explicat en la secció anterior. En la figura 2.3 es mostra un exemple en una secció 2D d'una xarxa vòxel.

2.4 Optimitzacions amb octrees

En el cas dels octrees no tenim el problema de perdre informació, ja que no transformem el propi point cloud. En aquest cas passem tant el point cloud com tots els nodes de l'octree, tant els nodes fulla com la resta. A més, el shader per trobar quin son els punts 'candidats' perquè formin part de la visualització el que fa es trobar els nodes fulla per on passa el raig. Com veurem en el següent capítol, aquests nodes fulla els troba ordenadament. És a dir, primer troba els nodes més proper a la càmera, i seguidament troba els nodes que no ha trobat ja que estiguin més a prop de la pantalla. Això és ideal per al nostre mètode, ja que significa que Q pot ser el conjunt de punts que hi ha al primer node fulla que troba que no estigui buit quan es faci la intersecció amb el raig con. Això també garanteix que el conjunt de punts que usem per al mètode estan tots en un entorn suficientment petit.

A més, aquesta construcció ens permet dir quin és el valor de t_{min} i t_{max} del mètode. Establim que t_{min} sigui el paràmetre tal que el raig xoca per primera vegada amb el node fulla, i, anàlogament, t_{max} com el paràmetre tal que el raig surt del node fulla trobat.

En la figura 2.4 veiem una secció 2D d'un octree donat i un exemple de com s'aplicaria el mètode presentat per a aquest cas. Veiem on es col·locaria la primera iteració de la x i on apuntarien els t_{min} i t_{max}.

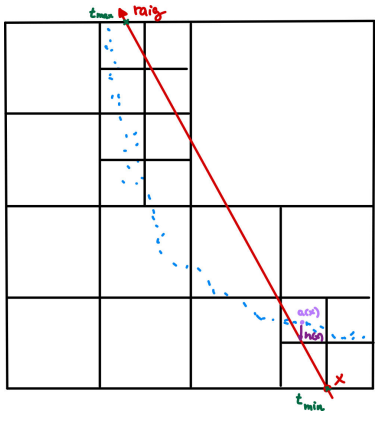

Figura 2.4: Exemple d'aplicació del mètode en un Octree. Secció 2D

Capítol 3

Disseny de l'aplicació

Per a la visualització del Point Clouds s'han usat dues arquitectures diferents: una basada en la CPU, i una aprofitant el potencial de la GPU. La primera arquitectura està completament basada en l'empleada a l'assignatura de Gràfics i Visualització de Dades. En canvi, per a la versió de GPU, em baso en l'arquitectura del programa fet pel Gerard Perelló Parello [13], la qual detallaré en aquest capítol.

Per a la visualització dels objectes usant la GPU, com hem vist anteriorment, hem de modelar, definir una estratègia de renderitzat, transferir les dades a la GPU i, finalment, executar la estrategia de visualització en la GPU. Per a aquest projecte, aquest pipeline s'ha modificat lleugerament:

CPU: Des de la CPU, es transforma les dades llegides des d'un fitxer perquè s'adaptin a les nostres configuracions, i perquè s'optimitzin per a la GPU. Aquests són els passos que segueix:

1. **Configuració, modelatge i entrada de dades:** Primer de tot s'estableixen les variables que defineixen el món, com per exemple saber on és la càmera, quin és el shader que volem usar, definim els materials, llums... Aquí també llegim des de fitxers l'objecte que volem, i inicialitzem el bucle de renderitzat i actualització. En poques paraules, configurem tot el que volem visualitzar i com ho volem veure.

2. **Construcció d'estructura de dades:** En aquest pas es construeixen estructures de dades eficients per a emmagatzemar els Point Clouds. En aquest treball s'usaran vòxels o octrees.

3. **Enviar les dades a la GPU**: En aquest pas s'estableixen com s'empaqueten i s'envien les dades per a cada una de les nostres estratègies. Aquest procés aprofita els buffers de connexió, que actuen com a intermediaris entre la CPU i la GPU, facilitant el flux fluid de les nostres dades.

GPU: Un cop arriba el point cloud a la GPU, es genera la imatge que veiem des de pantalla. Aquests són els passos que fa:

1. **Raytracing:** Profunditzarem en la nostra implementació completa del raytracing, explicant com es mostren els raigs, com hem dissenyat els nostres algorismes de visualització per a cadascuna de les nostres estructures de dades.

2. **Sortida:** Finalment, la imatge renderitzada s'envia al buffer de sortida per a la seva visualització a la pantalla.

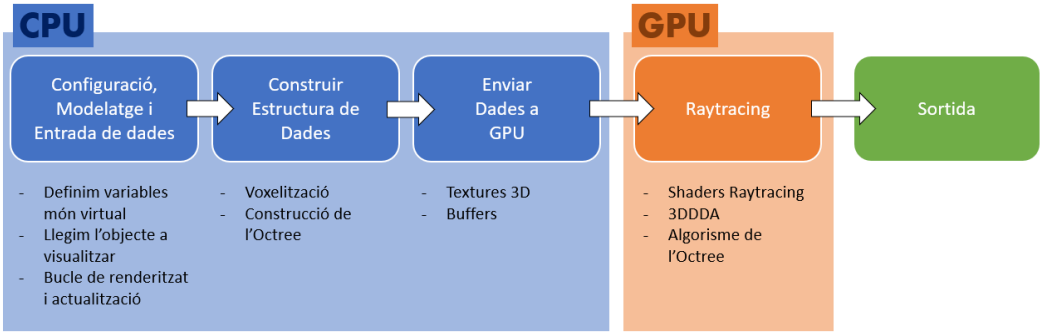

Figura 3.1: Esquema del Pipeline. Basat en l'esquema del TFG del Gerard Perelló [13]

Veiem en detall cada un dels passos.

3.1 Configuració, modelatge i entrada de dades

Com ja s'ha comentat, primer de tot s'ha d'establir quines són les variables que conformen l'escena que es volen veure. Per això hem de configurar els següents objectes:

1. **Càmera:** Per a la càmera especifiquem la seva posició en l'espai virtual (coord. càmara), el punt cap a on mira (look_at), el vector z-up, el qual determina l'orientació de la càmera, i la relació d'aspecte de la imatge. Amb aquests atributs podem crear la matriu de vista, i la de projecció, les quals serveixen per definir com es veurà la nostra escena des de la perspectiva de la càmera en el renderitzat de les imatges. Es mostren les variables en la figura 3.2.

2. **View Port:** En la tècnica de raytracing, es fa ús de z-buffer per activar el pipeline de la GPU. Aquest pipeline permet processar cada raig de llum en paral·lel, accelerant significativament el procés de renderització. Per aconseguir això, es prepara una malla que està adaptada per cobrir tota la pantalla. Aquesta malla està formada per quatre punts o dos triangles, tal com es mostra en la figura 3.3, que segueixen les directrius de les coordenades de dispositiu normalitzades (NDC). Les NDC, que són les sigles de Normalized Device Coordenates, són un sistema de coordenades estandarditzat on l'espai visible a la pantalla oscil·la entre -1 i 1 al llarg de cada eix, permetent definir la pantalla de manera uniforme. En lloc de fer servir mètodes tradicionals que depenen de transformacions de shader de vèrtexs utilitzant matrius

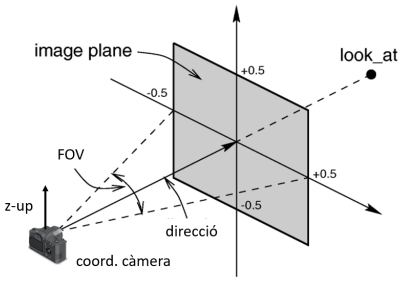

Figura 3.2: Variables de la càmera

View i Projection de z-buffer, aquest enfocament defineix directament quatre punts en l'espai clip. Això forma un quad que, després de la rasterització, cobreix completament la pantalla. Aquesta estratègia evita extenses transformacions, optimitzant el procés de renderització. Després de l'assemblatge de les primitives, aquestes estan preparades per a la rasterització. La tasca del rasteritzador consisteix a utilitzar els vèrtexs per formar primitives que abasteixen tota la pantalla. Cada punt individual que constitueix aquest quad, que està programat per a la representació com a píxel, s'identifica com un fragment. Notablement, com que els vèrtexs no tenen atributs més enllà de les seves posicions, el procés precedeix la interpolació d'atributs per complet. Això vol dir que no es necessita calcular cap atribut addicional per a cada vèrtex, el que fa que el procés sigui més eficient.

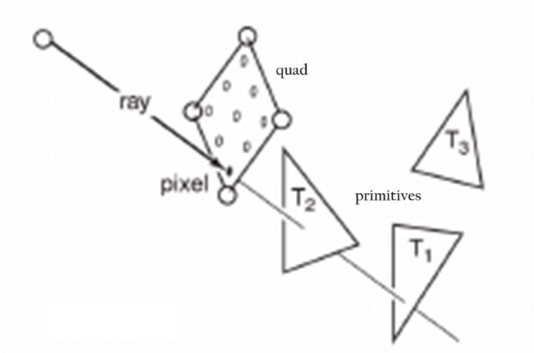

Figura 3.3: Representació del procés de raytracing a través del z-buffer. Figura modificada d'una imatge del article [15]

3. **Point Cloud:** També es determinen quin serà el point cloud que volem visualitzar, el qual podem llegir de fitxers obj, ply o txt, que són els formats desitjats (s'expliquen els formats obj i ply en la pàgina web de iDD [12]) . Es recopilen tots els punts del núvol de punts en una matriu de punts, on també s'emmagatzemarà qualsevol atribut extra que puguin tenir.

4. **Materials i Llums:** Inicialitzem materials i llums segons les estructures bàsiques explicades a Learn Open GL [7].

5. **Variables d'algorismes**: Finalment, també s'especifiquen les variables que modifiquen els mètodes d'optimització de Point Clouds (vegeu secció 1.3), o aquelles variables que varien els algorismes que usem per visualitzar-los (vegeu secció 2.2.1).

Ara, a més, hem d'inicialitzar el bucle principal de l'aplicació, la que permet veure a temps real els canvis que fem als atributs mencionats anteriorment. Aquest bucle està format pels següents dos mètodes: Actualització i Renderització.
Durant la fase d'**actualització** es realitzen actualitzacions de les variables GPU. Primer de tot, s'actualitzen totes les entrades, incloses les possibles tecles de teclat i el moviment del ratolí, en cas d'haver-hi canvis. Les variables també es poden actualitzar gràcies a la interfície gràfica del programa (vegeu figura 3.4). A continuació, es canvien les matrius, la lògica de visualització i les entrades a les memòries intermèdies de textura, perquè siguin les obtingudes a partir de les noves variables base.
Durant la fase de **renderització**, inicialment es processa la interfície gràfica de l'usuari. Després es renderitza cada un dels models que hem creat. Això implica la representació del viewport i del point cloud. En aquesta fase, els objectes s'envien com a dades, prioritzant la canonada z-buffer per a un únic objecte. Aquest procés implica la renderització i transmissió de la finestra de visualització. Les dades contingudes en les estructures del núvol de punts es combinen dins de la funció de representació de la finestra de visualització, ja que es tracta d'una analogia semblant a una textura.

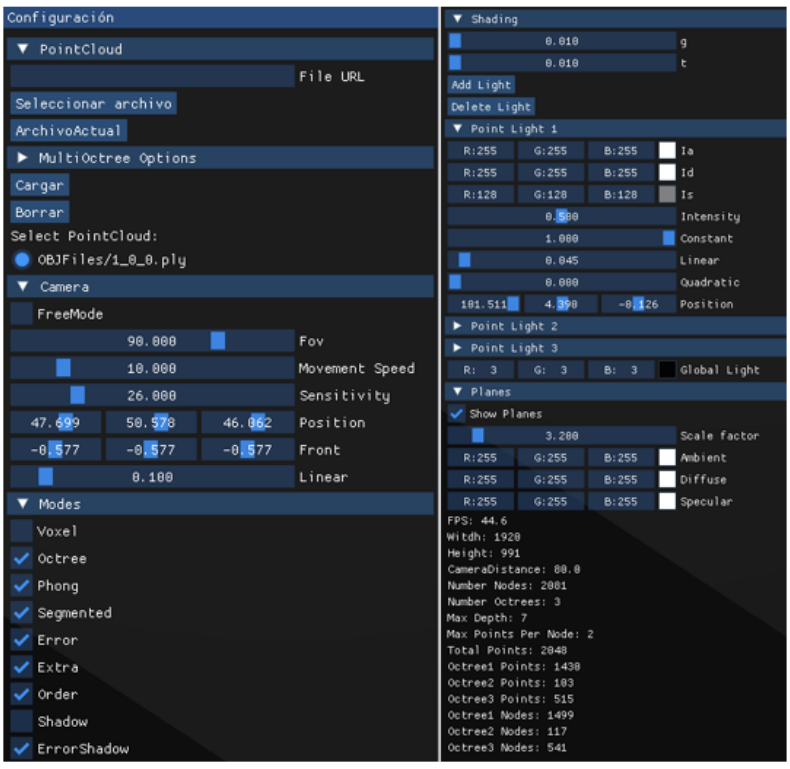

Figura 3.4: GUI del programa

3.2 Optimització

En aquest pas, hem d'optimitzar els point clouds perquè el nostre ordinador sigui capaç de processar-los. I això es pot aconseguir de dues formes diferents, com s'ha vist anteriorment: contenint el point cloud en un conjunt de **voxels** o contenint-lo en un **octree**.

3.2.1 Voxelització

Com s'ha descrit en la secció 1.3.1, voxelitzar un point cloud consisteix a agrupar els punts d'aquest en voxels d'una determinada mida. Això s'aconsegueix creant una "matriu de voxels", la qual és una matriu de 4 eixos. Però es pot veure com una matriu de 3 eixos amb les mateixes dimensions que la voxelització del point cloud, tal que cada element és el color del vòxel en aquella posició. El color és un vector de 4 coordenades: (R, G, B, α), on α representa la transparència. Per a poder crear aquesta matriu, el codi segueix els següents tres passos:

1. **initVariables():** En aquest pas es determinarà les variables per crear la matriu. Donada la mida del vòxel (un cub de costat b) i el nombre màxim de voxels per costat, transforma el point cloud perquè en voxelitzar-lo, tingui les característiques desitjades. A més, es calcula el nombre total de voxels (d'amplada s_x vòxels, alçada s_y vòxels i profunditat s_z vòxels).

2. **Inicialització de memòria per omplir les dades dels point clouds.** En aquest pas es crea la matriu de voxels. Es reserva en memòria l'espai que ocuparà, que és $O(s_x \times s_y \times s_z \times 4)$. Això es deu al fet que es guarda el color de cada vòxel, que està format per 4 ints.

3. **populateImage():** Un cop es sap quines són les coordenades de cada punt (i, per tant, a quin vòxel pertany), s'assigna el color del punt p_i al vòxel v_i a la seva coordenada corresponent (i, j, k) de la matriu, de manera que a la matriu es diferencia entre les coordenades que estan buides (aquelles amb valor (0, 0, 0, 0)) i les que tenen el valor del color.

Una vegada completats aquests 3 passos, la voxelització del point cloud està completa, ja que ja es té una matriu amb tots els vèrtexs discretitzats a la posició on els corresponen.

3.2.2 Construcció de l'Octree

Per a construir l'octree O que contindrà el nostre point cloud, s'aplica l'algorisme 1 recursivament per discretitzar l'espai del point cloud i afegir els punts al seu respectiu node de l'octree. Aquesta subdivisió segueix el criteri de subdivisió especificat en la secció 1.3.2. Així que cal tenir en compte els següents paràmetres:

1. **depthmax**. Aquesta variable determina el màxim nombre de subdivisions que podem fer a la caixa contenidora. Determina la mida de la subcapsa contenidora més petita de l'octree.

2. **pointsmin**. Determina el numero mínim de punts que ha de contenir una subcapsa perquè es continuïn fent subdivisions d'aquesta. Estableixen aquest llindar per controlar la granularitat de les nostres subdivisions. Més punts significa divisions més grans, menys precises, mentre que més punts donen com a resultat divisions més fines i més precises.

3. **sameLabelFlag**. Booleà que diu si tots els punts dins d'un node han de tenir el mateix tipus d'etiqueta. Això és útil quan les nostres dades del point cloud estan etiquetades i volem assegurar-nos que les subdivisions no separin dades de la mateixa etiqueta en diferents nodes.

Aquestes variables determinen quan un node de l'octree és una fulla o no. L'elecció i la calibració d'aquests criteris de subdivisió poden afectar significativament el rendiment final de l'algorisme, així que escollirem l'elecció que tingui millor rendiment, segons el point cloud escollit.

3.3 Enviar a GPU

A l'hora de treballar amb GPU, s'ha de tenir en compte que no pot manipular estructures de dades complexes amb tanta facilitat com a la CPU. Així que moltes vegades és necessari transformar les estructures de dades usades a la CPU a una forma que pugui ser processada per la GPU. I això es pot aconseguir de dues formes: usant textures o buffers. Una **textura** és un tipus especial de memòria intermèdia que s'utilitza per emmagatzemar dades d'imatge, i està particularment optimitzada per a patrons específics d'accés a la memòria. Les textures poden ser 1D, 2D o 3D. Aquestes últimes, en particular, són especialment bones per a emmagatzemar dades volumètriques, així que s'usaran per enviar la nostra estructura de vòxels a la GPU.
Finalment, un **buffer** és essencialment una regió d'emmagatzematge de memòria física utilitzada per emmagatzemar temporalment dades mentre s'està movent d'un lloc a un altre, i és el que s'usarà per enviar els octrees a la GPU.

3.3.1 Textura 3D per a la Voxelització

Una textura 3D és, bàsicament, una estructura de dades similar a un cub, amb cada vòxel dins del cub capaç d'emmagatzemar certa informació. Això coincideix totalment amb com hem emmagatzemat la voxelització del point cloud en la subsecció 3.2.1 així que es poden guardar directament qualsevol posició dins de la quadrícula per recuperar les seves dades associades. Aquestes dades seran el color per al vòxel específic. D'aquesta forma es poden transferir de manera eficient les nostres dades de la xarxa vòxel a la GPU de manera que sigui compatible amb OpenGL i estigui òptimament estructurada per al

processament de la GPU.

Cost en memòria: Si bé la textura 3D proporciona una manera eficient de transferir les nostres dades de la xarxa vòxel a la GPU, no és gaire eficaç per a l'ús de la memòria. L'espai ocupat és proporcional a les dimensions de la mateixa quadrícula (és $O(s_x \times s_y \times s_z \times 4)$, tal com hem especificat en la subsecció anterior). En particular, no treu avantatge sobre la distribució del point cloud. Per exemple, en escenaris amb point cloud molt dispers, aquesta complexitat podria precipitar una ineficiència de memòria significativa. Per a aquest motiu, també s'han considerat els octrees com una alternativa. Aquesta seria més eficient ja que, contràriament a la textura 3D uniforme, un octree només assigna memòria quan es requereix, és a dir, on se situen els punts del point cloud.

3.3.2 Buffers per l'Octree

Com es volen visualitzar directament els punts, no els punts discretitzats en vòxels, s'ha de prendre una estratègia més ambiciosa. En conseqüència, s'hauran d'enviar els nodes d'octree i els punts a la GPU mantenint una relació entre aquestes dues estructures, ja que és necessari saber com és aquest octree per a poder aplicar el raytracing. Com que no hi ha estructures de dades preestablertes a la GPU per a això, la nostra única opció és aplanar i sincronitzar ambdues estructures i enviar-les, tot respectant els següents criteris de sincronització:

1. Donat un node s'ha de poder trobar els nodes fills.
2. S'ha de poder saber la posició en l'espai virtual dels nodes i dels punts
3. Donat un node fulla s'ha de poder trobar els punts continguts en ell.

Per enviar els **punts** del point cloud, usarem una llista ordenada de **FlattenedPoints**'s, la qual anomenarem **FlattenedPoints**, tal com es descriu en la següent figura 3.5.

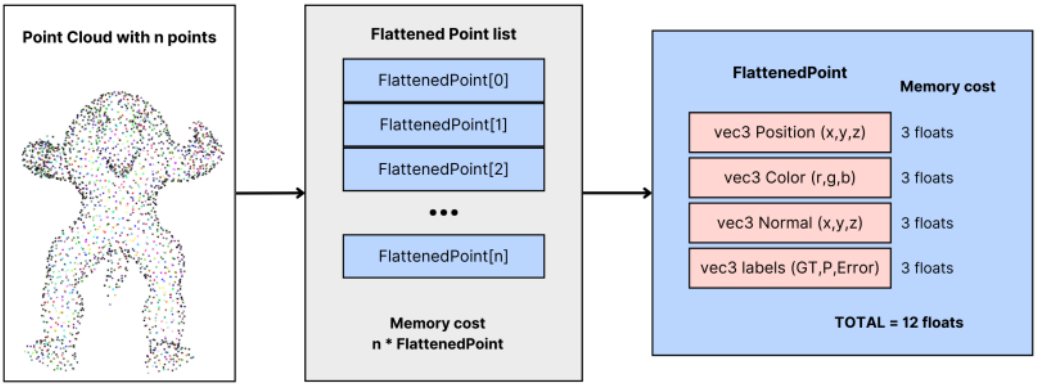

Figura 3.5: El procés de transformació d'un point cloud amb n punts, a una llista de n **FlattenedPoint**'s. Descripció de l'estructura d'un **FlattenedPoint**. Figura extreta del TFG del Gerard Perelló [13].

I per enviar els nodes, usarem una llista ordenada de **FlattenedNode**'s, la qual anomenarem **FlattenedNodes**, tal com es descriu en la següent figura: Aquest vector ha de

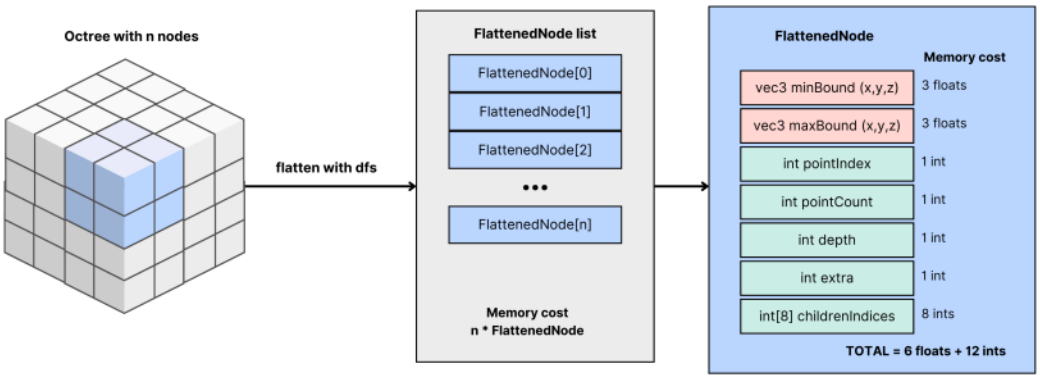

Figura 3.6: El procés de transformació d'un octree amb n nodes, a una llista de n **FlattenedNode**'s. Descripció de l'estructura d'un **FlattenedNode**. Figura extreta del TFG del Gerard Perelló pere[13].

mantenir l'estructura jeràrquica de l'octree en un format lineal òptim per al processament de la GPU. Per fer-ho, transformarem l'octree usant l'algorisme Depth-First Search (DFS), explicat en diverses assignatures de la carrera. I quan s'apliqui DFS, es poblarà la llista **FlattenedPoints** amb punts dels nodes de les fulles. Al mateix temps, capturem els atributs de cada node descrits a la figura 3.6 per a poblar l'estructura de llista **FlattenedNode**. Com es desa l'índex de cada node fill dins de la llista childrenIndices del **FlattenedNode** de cada node, no és necessari mantenir l'orde dels nodes, fent que es pugui accedir ràpidament als fills de cada node.
I per al **FlattenedPoints** necessitarem mantenir un ordre "local". Això és perquè volem trobar ràpidament tots els punts que es troben dins d'un node fulla. Per això, cada **FlattenedNode** té els atributs pointIndex i pointCount, els quals determinarien on són tots el punts continguts en el node si estiguessin tots agrupats seqüencialment en la llista. Ara bé, la posició d'aquest grup dins de la llista no és important, i per això l'ordenació es local. Una vegada hem completat el DFS, podrem enviar les llistes **FlattenedPoints** i **FlattenedNodes** a la GPU a través de buffers, cadascun definit per la seva mida apropiada.

Quant al cost de la memòria, s'envien dues llistes a la GPU. Les dimensions dels elements d'aquestes llistes varien. Els elements de la llista **FlattenedPoints** són **FlattenedPoint**'s i ocupen 12 floats cadascú. En canvi, els element de la llista **FlattenedNodes** són **FlattenedNodes**'s, els quals estan composts per 6 floats i 12 ints cadascú. Assumint que la mida ocupada pels ints és equivalent a la dels floats, i denotant n, m com el nombre de punts i el nombre de nodes de l'octree, respectivament, es té que el cost total de memòria es aproximadament $O(12n + 18m)$. Aquesta estimació ofereix una perspectiva general del cost de memòria. No obstant això, la realitat pot variar segons la densitat dels punts a l'espai i els criteris de subdivisió de l'octree.

3.4 Raytracing a GPU: descripció dels shaders

Com s'envien diferents estructures depenent de l'optimització que usem per al point cloud que volem usar, hem de tenir diferents shaders per a cada una de les estratègies d'optimització. Per tant, s'ha dissenyat un shader per a poder renderitzar el point cloud **voxelitzat** i un altre per a poder renderitzar el point cloud contingut en un **octree**. Per al primer usarem l'estratègia de 3DDDA, explicada més endavant. En canvi, pel segon shader farem servir un DFS per poder trobar tots els nodes fills que travessa el raig. Cal destacar que per a cada estratègia també s'usarà un shader diferent. Però en tots els casos seguiran la mateixa lògica base per a poder trobar el conjunt de punts "candidats" per a poder xocar amb el raig.

3.4.1 Base dels shaders

Si es vol fer ray sampling en el fragment shader, és necessari mostrejar un raig per a cada píxel de la pantalla. Així que primer de tot crearem les coordenades 2D de la nostra pantalla (u, v). Tot seguit, usant les matrius View i Projection descrites en detall en el treball del Gerard Perelló [13], podem obtenir cada una de les direccions dels raigs que surten de la càmera, i per tant aconseguim el conjunt de raigs que s'usen per al renderitzat.
Un cop trobem els raigs, creem un mètode, *hit* per a cada un dels shaders, és a dir, el mètode que detectarà si el raig col·lisiona amb el point cloud i, en cas de fer-ho, retorni el punt on ha xocat, la normal del point cloud en aquell punt, el color, entre altres... Aquest mètode, primer de tot, troba caixes contenidores més petites on es pot trobar la col·lisió. Tot seguit, aplica l'algorisme de visualització del point cloud corresponent (en el nostre cas, o l'algorisme de les esferes o el definit en la secció 2.2), i finalment retorna els atributs descrits abans.

3.4.2 Shader Voxel: 3D Digital Differential Analyzer (3DDDA)

En primer lloc, veiem l'estratègia a seguir quan s'obtingui l'estructura que correspongui a la textura 3D, és a dir, la voxelització del point cloud. La primera aproximació utilitzada per visualitzar el point cloud és l'ús de l'algorisme DDA. Així que, donat un raig, la idea és utilitzar la posició inicial o i la direcció $\vec{d}$ del raig per a poder saber quins vòxels es visitaran i en quin ordre. L'algorisme té com a objectiu minimitzar el nombre de vòxels visitats al llarg de la línia utilitzant un sistema de prioritat per a les coordenades, ordenat pel paràmetre t, que mesura les distàncies d'intersecció al llarg de la línia.

Inicialment, calculem els valors t_{min} i t_{max} per a la capsa contenidora general de la xarxa. Si no interseca, retornem false. En cas contrari, travessarem els vòxels, i calcularem els valors t pels quals el raig interseca cada voxel travessat, els quals anomenarem t_{next}. La base de l'algorisme és el procés en bucle on, a cada iteració, identifiquem el valor més petit del t_{next}. Aquest valor indica el límit del voxel que el raig trobarà a continuació. En conseqüència, avancem cap al següent vòxel que s'interseca al llarg del camí del raig.

Aquest procés iteratiu persisteix fins que localitzem un voxel que compleix la condició d'encert especificada (si l'algorisme de visualització ho determina) o fins que sortim de la caixa contenidora del volum. Fixem-nos en l'exemple de la figura 3.7. El punt negre representa el t_{min} i el blanc el t_{max}, ja que són els punts on el raig interseca la capsa contenidora de la xarxa de vòxels. Tot seguit, calculem els valors de t_{next}, que són aquells on el raig creua un pla que divideix la xarxa. En aquest cas cada punt es la intersecció del raig amb el pla amb el mateix color. Així que en aquest cas el raig travessa 4 vòxels.

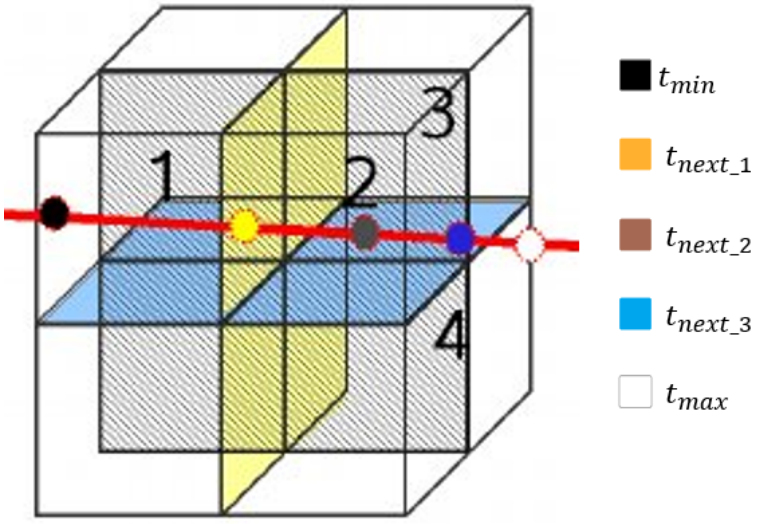

Figura 3.7: Exemple d'aplicació 3DDDA

L'eficiència d'aquest algorisme resideix en el fet que només visita aquells vòxels que travessa, així que no fa visites redundants.

3.4.3 Algorisme d'Octree

Per veure quins són els nodes que interseca el raig, primer de tot es comprova si el raig travessa el Octree en general (és a dir, la capsa contenidora del point cloud). En cas que així sigui, aplicarem la funció `dfs_stack_rt()` de l'algorisme 3.

Aquesta funció és un DFS que va trobant els nodes pels quals el raig travessa. Si el node trobat és un node fulla, aplica la funció de `leafOperation(currentnode)`, la qual serveix per aplicar l'algorisme de visualització per als punts dins d'aquest node fulla. En cas que no sigui una fulla, apliquem la funció `get_intersect_nodes_rt`.Aquesta el que fa és primer de determinar l'ordre de visita dels nodes fills en funció de la direcció del raig. Això s'aconsegueix mitjançant la funció auxiliar `GetVisitOrder()` i optimitza la travessa prioritzant els nodes més propensos a intersecar amb el raig utilitzant la direcció del raig. Els nodes fills s'afegeixen a la pila si el raig interseca les seves caixes contenidores en qualsevol instant. Finalment, es retorna la llista ordenada de nodes a visitar.

Algorithm 3 Travessa de l'Octree DFS per al RT

Input: Node root de l'Octree: *root*, Raig: *ray*, t mínima per al RT: t_{min}, t màxima per al RT: t_{max}, HitInfo per al RT: *hit*

```
1:  function DFS_STACK_RT(root, ray, t_min, t_max, hit)
2:      stack ← new Stack();
3:      stack.push(root);
4:      while not stack.isEmpty() do
5:          currentNode ← Stack.pop();
6:          if currentNode.leaf() then
7:              leafOperation(currentNode);
8:          else
9:              selectedNodes ← get_intersect_nodes_RT(currentNode, ray, t_min, t_max);
10:             for each node in selectedNodes do
11:                 Stack.push(node);
12: function GET_INTERSECT_NODES_RT(currentNode, ray, t_min, t_max)
13:     int visitOrder[8];
14:     Node nodesToVisit[];
15:     visitOrder ← GetVisitOrder(ray.d⃗);
16:     for i in 7 downto 0 do
17:         childIndex ← visitOrder[i];
18:         childNodeIndex ← currentNode.childrenInfo[childIndex];
19:         childNode ← getNode(childNodeIndex);
20:         if rayIntersectsNode(ray, childNode, t_min, t_max, ϵ) then
21:             nodesToVisit.add(childNode);
22:     return nodesToVisit;
```

Aquest algorisme permet trobar els punts ordenats per grups de més propers als més llunyans. Així que aquest mètode va bé per a tots els algorismes que requereixin només els punts més propers a la càmera per fer la renderització. Tot i això, es poden fer variacions a l'algorisme, perquè es tinguin en compte tots els punts dins de tots els nodes fulla que travessa el raig. En el nostre cas, aquest algorisme ens afavoreix, ja que el nostre mètode de visualització només necessita un petit entorn de punts per calcular la superfície implícita.

També cal destacar que, com en el cas dels vòxels, només es visiten aquells nodes fulla que intersequen el raig, i per tant és eficient.

Capítol 4

Simulacions i resultats

En aquest capítol es veurà la comparació entre el mètode de visualització que ha usat el Gerard Perelló en el seu TFG(el mètode de punts esferes) i el mètode explicat en aquest treball. Compararem tant el rendiment com els resultats obtinguts a les visualitzacions.

4.1 Especificacions

A l'hora d'experimentar i veure simulacions, s'ha de tenir en compte l'entorn on s'executen. Per tant, és necessari dir en quin context executaré el codi. S'han utilitzat dos ordinadors diferents, un per a executar el codi de la CPU i un altre per executar el codi de la GPU.

CPU:

- **Sistema operatiu:** Ubuntu
- **Model de sistema:** Lenovo idepad 330
- **RAM:** 8 GB DDR4 SDRAM.
- **CPU:** Intel Core i5 (8a generació) 8250U.
- **Velocitat de rellotge:** 1.6 GHz.
- **Disc dur:** 256 GB SSD.

GPU:

- **Sistema operatiu:** Windows 11 Pro versió 22621.3007
- **Model de sistema:** MSI Katana GF66 11UD.
- **Targeta gràfica:** NVIDIA GeForce RTX 3050 Ti.

- **RAM:** DDR IV 8GB*2 (3200MHz).
- **CPU:** Intel Core i7-11800H.
- **Disc dur:** 1TB NVMe PCIe Gen3x4 SSD.

4.2 Metodologia

Per a aquest experiment, primer de tot, mirarem com es comporta el mètode tant quan usem només CPU com quan també aprofitem la GPU.

Visualitzarem, en ambdós casos, el point cloud emmagatzemat en el fitxer `1_0_0.ply` per fer les comparacions. Aquest conté 2048 punts, i venia amb el codi que va crear el Gerard Perelló, que ell mateix explica el seu origen en el seu treball [13].

Com a la CPU no hi ha mètodes d'optimització, compararem allà les diferències entre les dues funcions de pes descrites en la secció 2.2.1. I pel programa que usa GPU, veurem les diferències entre usar voxelització o octrees per optimitzar la visualització, establint la funció de pes com $\theta(x) = \frac{1}{x+0.001}$ únicament. Per a cada un dels casos descrits, veurem com es comporten segons la tolerància ϵ i la γ establerts. En la GUI de la GPU, per al octree, es poden modificar les variables t i g. Tal com s'ha detallat en la secció 2.2.1, tenim que $t = -log_{10}(\epsilon)$ i que $g = -log_{10}(1-\gamma)$. La figura 4.1 resumeix els casos que es provaran en les diferents seccions.

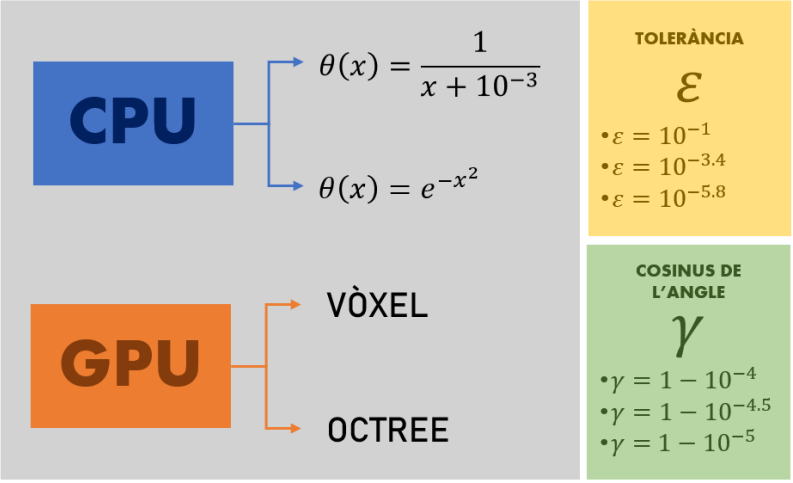

Figura 4.1: L'experiment 1 constarà de 4 casos principals, emmarcats en gris a l'esquerra. Per a cada cas, mirarem els 9 subcasos que sorgeixen a l'agafar una tolerància i una γ dels presentats aquí. Tenim que $t = 1, 3.4, 5.8$ i que $g = 4, 4.5, 5$.

Finalment, veurem els resultats finals en diferents casos puntuals, establint diverses variacions. Per exemple, veurem què passa quan la gamma és molt petita o quan visualitzem altres point clouds.

4.2.1 Mètriques

En tots els casos es mostraran imatges veient com és visualitza el point cloud. A partir d'aquí, cada cas principal, pel que fa a les especificacions, s'avaluaran diferents mètriques.

En el cas de la CPU mirarem el **temps que tarda en visualitzar**, en segons, i avaluarem el **nombre d'iteracions** necessàries per convergir en cada cas.

En el cas del cas de la GPU, tindrem en compte els **FPS (fotogrames per segon)**, el qual és una mesura estàndard del rendiment gràfic. Un FPS més alt implica una visualització més suau i fluida.

4.3 Resultats

Veiem llavors els resultats obtinguts. **En la següent secció es farà la interpretació de les dades obtingudes.** Comencem veient el cas principal de la CPU i després el cas de la GPU.

4.3.1 CPU

Primer de tot es mostraran les simulacions obtingudes a la CPU, amb el primer ordinador. Començarem provant amb la funció de pes $\theta(x) = \frac{1}{x+10^{-3}}$, i, tot seguit, la funció $\theta(x) = e^{-x^2}$.

Cas $\theta(x) = \frac{1}{x+10^{-3}}$

t \ g	**4**	**4.5**	**5**
1	7.059559s	6.838590s	7.023756s
3.4	7.187857s	8.332468s	7.910422s
5.8	7.791465s	7.006748s	7.083457s

Taula 4.1: Temps de renderització segons t i g usant $\theta(x) = \frac{1}{x+10^{-3}}$.

Per a t=1, la majoria dels hits es fan en 2 iteracions. Per a t=3.4, la mitjana d'iteracions puja a 4 aproximadament, havent-hi alguns raigs que convergeixen en més de 20 iteracions. I per a t=5.8, la majoria requereixen una mitjana aproximada de 10 iteracions. Veiem que s'obté en mitja un frame cada 7 segons aproximadament, lluny del temps real que voldríem.

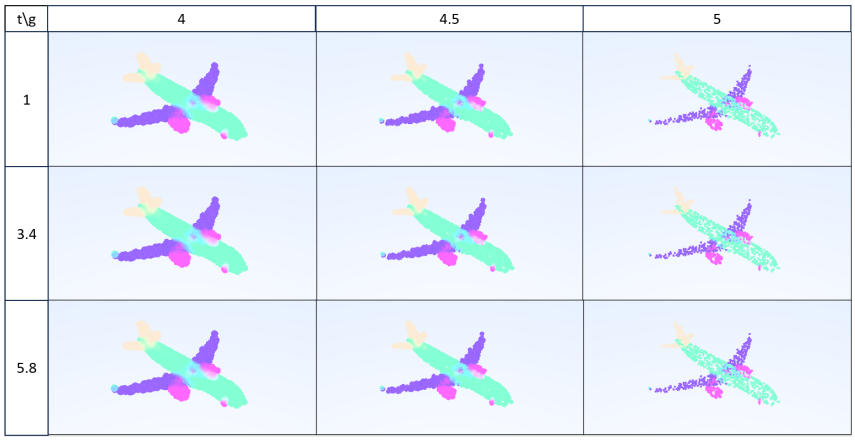

Figura 4.2: Visualització en CPU segons t i g usant $\theta(x) = \frac{1}{x+10^{-3}}$.

Cas $\theta(x) = e^{-x^2}$

t \ g	**4**	**4.5**	**5**
1	6.771618s	6.803126s	6.808279s
3.4	7.145594s	7.105734s	6.847301s
5.8	7.486342s	7.149955s	7.006085s

Taula 4.2: Temps de renderització segons t i g usant $\theta(x) = e^{-x^2}$.

Per a t=1, la majoria dels hits es fan en 2 iteracions, igual que en el cas anterior. Per a t=3.4 i t=5.8, la majoria requereixen una mitjana aproximada de 3 iteracions. I observem que s'obté en mitja un frame cada 7 segons aproximadament, una mica menys que en el cas anterior però lluny del desitjat.

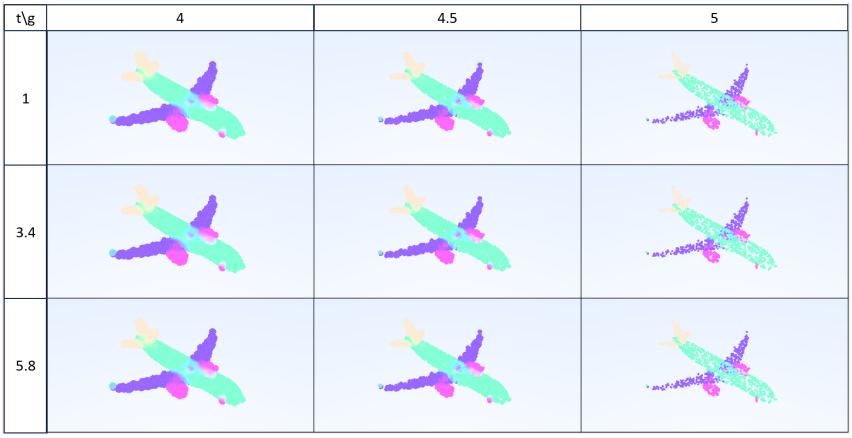

Figura 4.3: Visualització en CPU segons t i g usant $\theta(x) = e^{-x^2}$.

4.3.2 GPU

A continuació es mostren les simulacions obtingudes a la GPU, amb el segon ordinador. Primer provarem el model de vòxels primer, i després el model d'octree. La interpretació dels resultats es farà en la següent secció.

Vòxels

t \ g	4	4.5	5
1	93 FPS	90 FPS	56 FPS
3.4	55 FPS	44 FPS	56 FPS
5.8	17 FPS	18 FPS	19 FPS

Taula 4.3: FPS segons t i g en la voxelització.

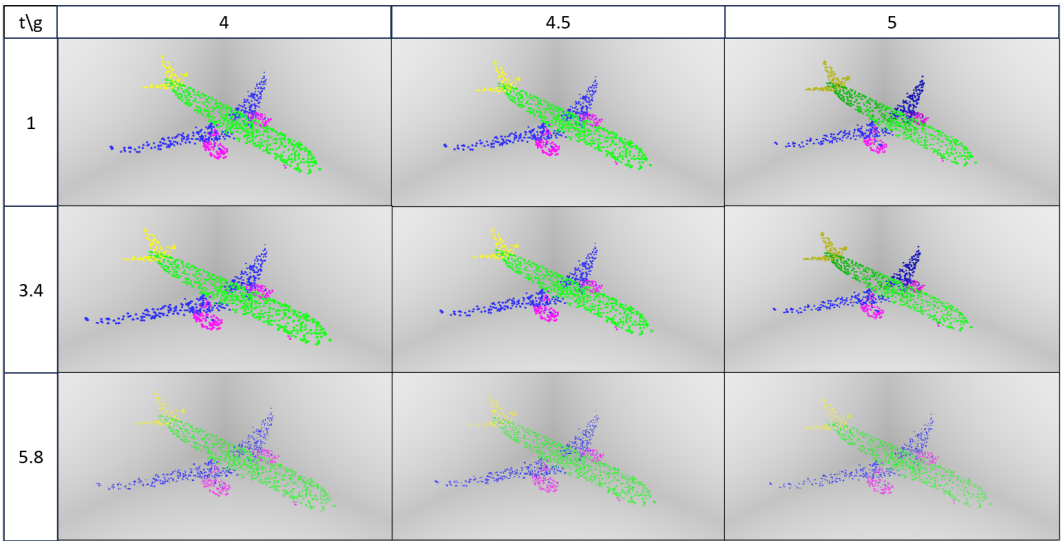

Figura 4.4: Visualització en GPU segons t i g en la voxelització.

Octree

t \ g	4	4.5	5
1	70 FPS	60 FPS	54 FPS
3.4	78 FPS	62.5 FPS	54 FPS
5.8	28 FPS	36 FPS	45 FPS

Taula 4.4: FPS segons t i g usant un octree.

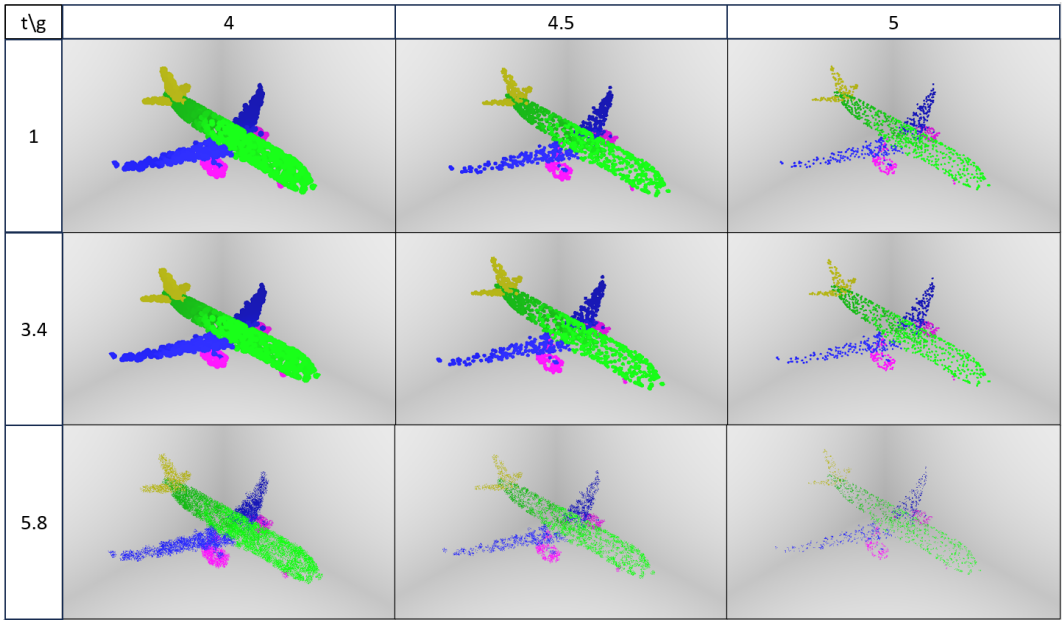

Figura 4.5: Visualització en GPU segons t i g usant un octree.

4.3.3 Punts esfera

En aquesta secció es mostren els resultats obtinguts amb el mètode de punts esferes usat en el TFG del Gerard Perelló [13]. S'ha tornat a córrer en el mateix ordinador que els resultats mostrats en la secció de la GPU a fi de tenir validacions comparables.

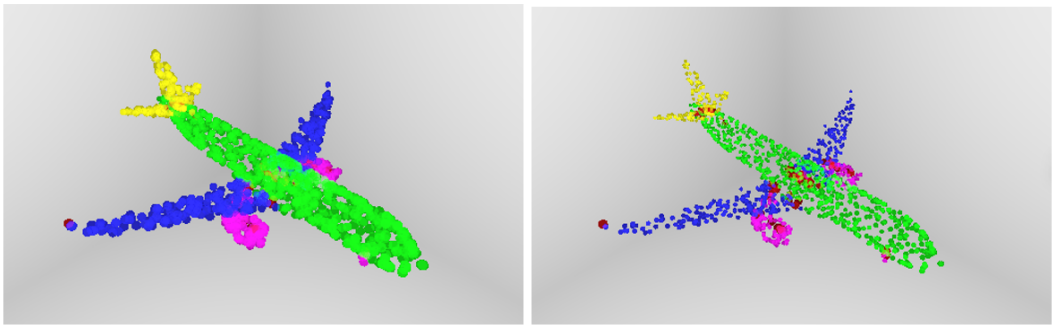

Figura 4.6: Dos exemples de visualització de l'avió usant el mètode de punts esfera. Fa servir una estructura Octree. Visualitzat gràcies al codi del Gerard Perelló.

I obtenim que per a qualsevol radi que els hi apliquis als punts, el point cloud renderitza en un promig de 10 FPS. Podem observar que la imatge de l'esquerra de la figura 4.6 s'assimila a quan visualitzem el nostre mètode amb octrees i $t = 1, g = 4.5$. I la imatge de la dreta de la figura s'assembla a quan apliquem el nostre mètode amb octrees i $t = 1, g = 5$.

4.4 Interpretació dels resultats

Primer de tot mirem la comparació entre les dues funcions de pes mostrades en la CPU. Podem veure que, en general, la visualització no canvia. Tot i això, usant la gaussiana es fan menys iteracions, i, en general, tarda menys en renderitzar. Ens podem fixar, però, que la diferència no és tan alta. Això es deu al fet que calcular l'exponencial és més costós que calcular la funció $\frac{1}{x}$.

També ens podem fixar en el fet que, en el cas de la CPU, no canvia gaire la visualització quan canviem la tolerància del mètode. Tot i això, augmenta el temps de renderitzat. Per tant, veiem que la convergència del mètode és ràpida, com ja intuíem.

Fixem-nos en la GPU. Per la naturalesa dels vòxels, no s'aprecien diferències notables per les g's escollides. Això es deu al fet que si el raig no interseca amb el vòxel físic, no considera que el punt forma part dins del con.

A diferència de la CPU, tant en els vòxels com en l'octree es pot observar que, per a una tolerància molt petita, en alguns raigs el mètode no convergeix quan amb toleràncies més petites si ho feia. Això es deu al fet que passa de les 100 interacions. I, tot i que no hi ha diferències en la visualització quan renderitzem amb $t = 1$ en comparació a quan renderitzem amb $t = 3.4$, si veiem que baixa significativament el rendiment. També ens fixem que el rendiment es redueix bastant també quan es posa $t = 5.6$. També, quan $t = 1, 3.4$, el rendiment es redueix quan augmentem g. Però quan $t = 5.8$, augmenta quan augmentem g. Això passa perquè, com ara en cada píxel que xoca contra punts es fan un munt d'iteracions, com menys vegades 'xoqui', menys càlculs es faran.

Finalment, podem observar com, per a t=1 és més eficient aplicar voxelització que octrees. En canvi, per a t=3.4 i t=5, és més eficient aplicar octrees que voxelització. A més, l'octree es més estable a variacions de tolerància o gamma que els vòxels.

A més, comparat amb el mètode dels punts esferes, és més eficient el mètode de Newton que s'ha estudiat en aquest treball. Veiem que, per al nostre mètode podem obtenir les mateixes visualitzacions variant la g segons el radi desitjat i establint $t = 1$. En canvi, els punts esfera obtenen 10 FPS a diferència dels 60 FPS que obtenim de promig amb el mètode iteratiu.

Com a apunt final, en el projecte de GitHub [1] hi ha una carpeta amb les visualitzacions de la secció anterior amb més detall.

4.5 Altres Visualitzacions

Per finalitzar el capítol, s'ensenyaran algunes captures del programa per aprofundir en el funcionament del codi.

Comencem amb la figura 4.7. Es mostra en detall com es veu el point cloud quan fem zoom.

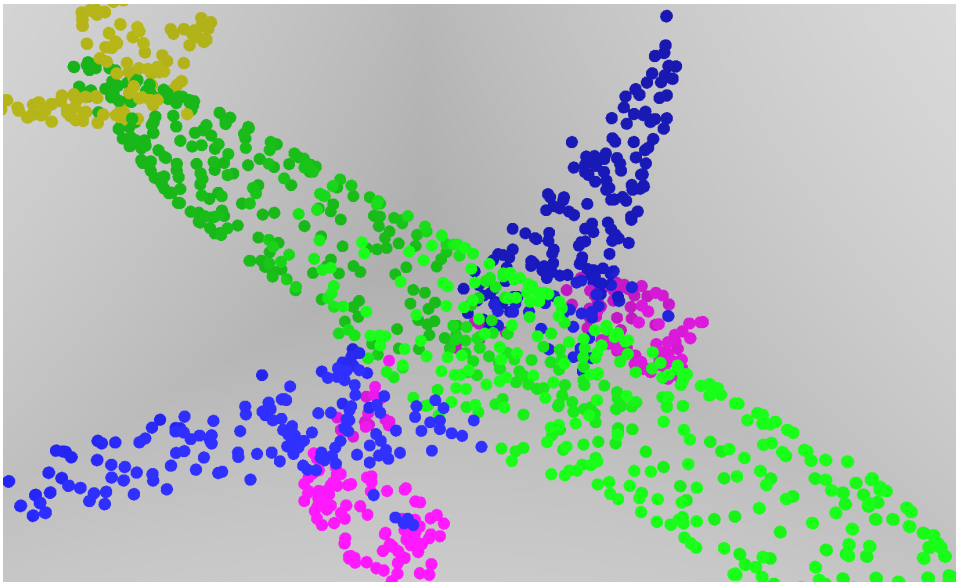

Figura 4.7: Visualització en detall dels 'punts' quan apliquem octree.

En la figura 4.8 veiem com, si posem una g molt petita, podem apreciar els nodes fulla de l'octree que conté el point cloud. Això es deu al fet que només revisem els nodes que creua el raig, i no tots els possibles nodes que continguin punts 'candidats' a ser renderitzats.

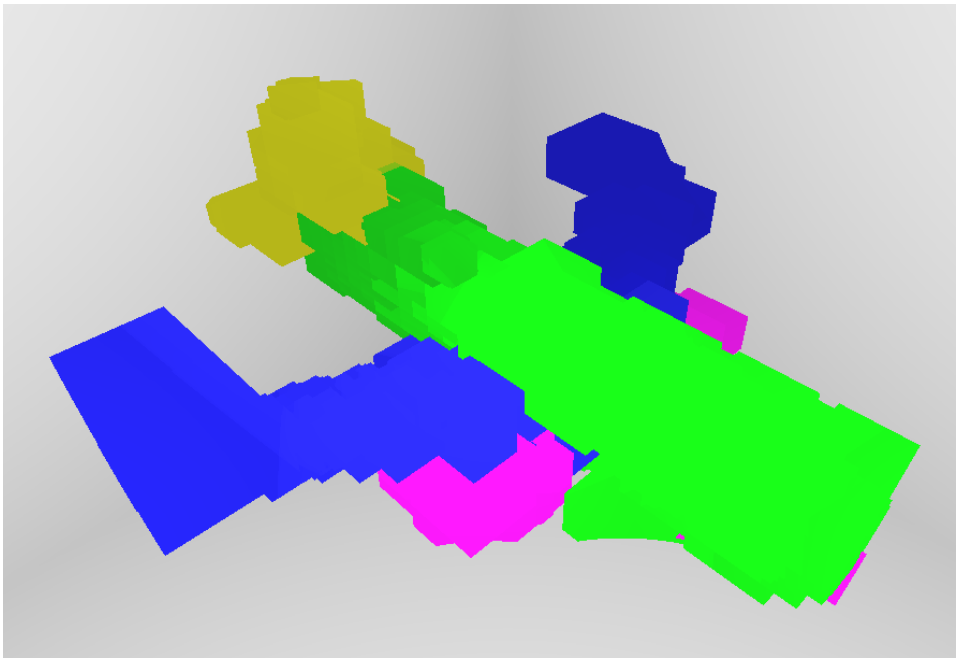

Figura 4.8: Nodes fulla de l'octree que conté el point cloud.

En aquesta tercera figura, la 4.9, veiem com es veu un altre point cloud al aplicar el mètode i usant octrees com a mètode d'optimització.

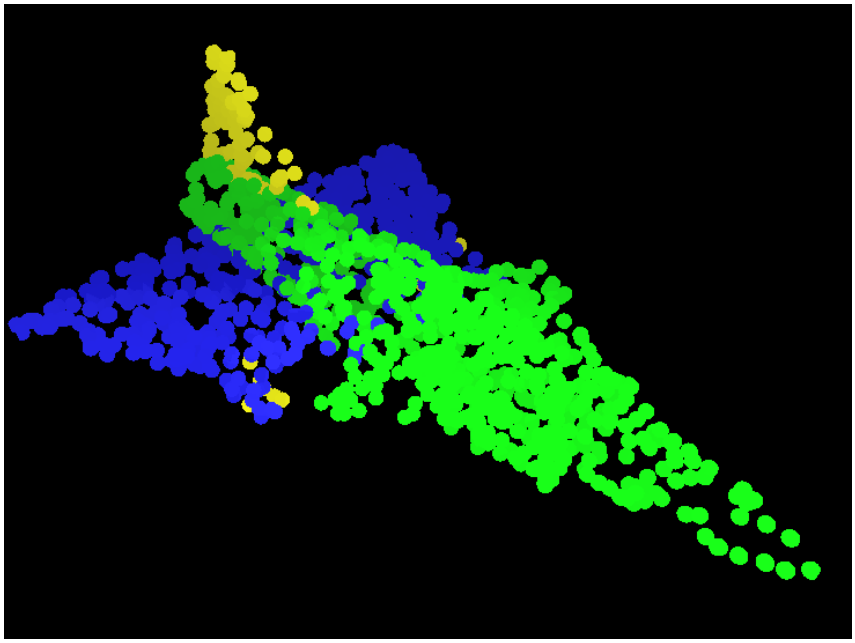

Figura 4.9: Visualització del point cloud en el fitxer `2197_21_4.ply`.

Finalment, veiem com es veuen els vòxels des de diferents angles en la figura 4.10. Des d'amunt es veuen com caixes, quan estem al mateix nivell les veiem com semiesferes i quan les mirem des d'avall semblen quarts d'esferes. Això es deu al fet que es fa la intersecció entre el con i un vòxel. Quan aquest con, quan arriba al vòxel, té un radi més gran, es veu el vòxel sencer. En canvi, si és més petit, es poden veure les interseccions entre el vòxel i els raigs cons (que conjuntament formen un con).

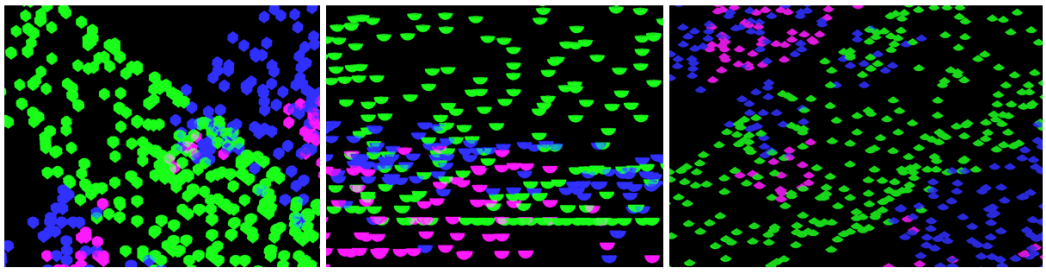

Figura 4.10: Els vòxels en detall vists des d'amunt (esquerra), al mateix nivell (mig) i des d'avall (dreta).

Capítol 5

Conclusions i feina futura

5.1 Conclusions del TFG

Després de fer aquest Treball de Fi de Grau (TFG), s'ha aprofundit un coneixement significatiu sobre diversos aspectes importants en el camp de la visualització de point clouds. En primer lloc, s'ha analitzat una solució no trivial de la visualització de point clouds, per determinar i definir bé quin problema tenim a resoldre quan es vol visualitzar un point cloud i com es pot representar virtualment.

En segon lloc, s'han explorat les tècniques de voxelització i octrees, comprenent com aquestes estructures de dades poden ser utilitzades per a optimitzar la visualització eficient de grans point clouds.

Aquesta investigació també ha dut a estudiar diversos mètodes de visualització, com ara splatting, punts esfèrics i raigs cònics/cilíndrics. Aquesta diversitat de tècniques ha proporcionat una visió àmplia de les opcions disponibles i les seves aplicacions pràctiques.

Un dels punts fonamentals d'aquest treball ha estat l'estudi del mètode iteratiu de Newton modelitzant els raigs com cons per visualitzar els point clouds. S'han estudiat els seus fonaments, i s'han codificat tant per a una arquitectura que només usa la CPU, com per una arquitectura que s'aprofita de la potència de la GPU.

A més, s'ha realitzat el disseny de l'aplicació que he fet servir com a base per a visualitzar els points clouds. S'han implementat tècniques d'optimització, com ara la voxelització o la creació d'un octree, perquè es puguin passar eficientment les dades llegides des de la CPU a la GPU, perquè aquesta pugui processar-les.

Els resultats obtinguts mitjançant el mètode iteratiu de visualització han estat analitzats, comparant cada un dels casos per veure quin és més eficient o quin mostra millor el point cloud. Hem pogut comprovar que el mètode estudiat en aquest treball és més eficient que el mètode de punts esferes, el qual era el mètode usat en el codi original creat per en Gerard Perelló, tot i que en algunes visualitzacions es produeixen alguns artefactes addicionals que cal matisar configurant els paràmetres del mètode.

En resum, aquest TFG ha proporcionat una visió integral dels mètodes de visualització de points clouds, i ha profunditzat en els conceptes de voxelització i octrees, i com es passen aquestes estructures a GPU. A més, s'ha estudiat un mètode en concret, del qual hem vist com és d'eficient.

5.2 Feina futura

Aquest treball estableix una alternativa a les visualitzacions estudiades en el treball d'en Gerard Perelló [13]. Tot i això, encara hi ha molta feina per fer:

Els codis poden ser optimitzats perquè es pugui visualitzar més eficientment els point clouds. A més, es poden considerar les altres possibles estratègies de voxelització explicades en la secció 2.3. Es pot estudiar en més profunditat quina funció de pes és la més eficient per a la visualització. O es pot veure quina tècnica d'optimització de dades pot aportar més eficiència a la visualització, comparant l'octree i la voxelització amb altres mètodes, com per exemple els kd-trees. O, fins i tot, es pot comparar l'estratègia dels punts esferes i el mètode estudiat amb el splatting.

Apèndix A

Manual tècnic

A.1 Requeriments

Els requeriments mínims per poder assegurar que es pot instal·lar el programari estan especificats en la secció 4.1

A.2 Com instal·lar el software

Els projectes que s'han usat estan penjats en els repositoris de GIT [1] i [2]. En els README.md estan les instruccions per instal·lar els programes. El primer repositori s'executa en Visual Studio i és el codi que usa la GPU. El segon s'executa en QT Creator i es el que usa únicament la CPU.

Bibliografia

[1] GitHub - tfg_metode_newton. https://github.com/SFPOL/TFG_Metode_Newton. [Accessed 17-01-2024].

[2] GitHub - tfg_metode_newton_cpu. https://github.com/SFPOL/TFG_Metode_Newton_CPU. [Accessed 17-01-2024].

[3] Revolutionizing Autonomous Vehicles: Unveiling the Power of Point Clouds — linkedin.com. https://www.linkedin.com/pulse/revolutionizing-autonomous-vehicles-unveiling-power-point-talha/. [Accessed 17-01-2024].

[4] Bart Adams, Richard Keiser, Mark Pauly, Leonidas J. Guibas, Markus Gross, and Philip Dutré. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum*, 24(3):677–684, 2005.

[5] Anders Adamson and Marc Alexa. Approximating and Intersecting Surfaces from Points. https://graphics.stanford.edu/courses/cs468-03-fall/Papers/alexa_surfintersect.pdf. [Accessed 07-01-2024].

[6] Thomas Leimkühler George Drettakis Bernhard Kerbl, Georgios Kopanas. 3D Gaussian Splatting for Real-Time Radiance Field Rendering — repo-sam.inria.fr. https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/. [Accessed 13-01-2024].

[7] Joey de Vries. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL — learnopengl.com. https://learnopengl.com/. [Accessed 17-01-2024].

[8] Dylan Ebert. Introduction to 3D Gaussian Splatting — huggingface.co. https://huggingface.co/blog/gaussian-splatting, 2023. [Accessed 12-01-2024].

[9] Erik Hubo, Tom Mertens, Tom Haber, and Philippe Bekaert. The quantized kd-tree: Efficient ray tracing of compressed point clouds. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 105–113, 2006.

[10] Petrus Kiv. RAY TRACING METHODS FOR POINT CLOUD RENDERING. https://trepo.tuni.fi/bitstream/handle/10024/117953/KiviPetrus.pdf#page=66&zoom=100,150,942. [Accessed 04-01-2024].

[11] M. Levoy and T. Whitted. The Use of Points as a Display Primitive — graphics.stanford.edu. https://graphics.stanford.edu/papers/points/. [Accessed 05-01-2024].

[12] Alessandra Paras. Understanding STL, PLY, and OBJ Files in Digital Dentistry - Institute of Digital Dentistry — instituteofdigitaldentistry.com. https://instituteofdigitaldentistry.com/3d-printing/understanding-stl-ply-obj-files-in-digital-dentistry/. [Accessed 02-01-2024].

[13] Gerard Perelló Martínez. Interactive multiple point cloud rendering using gpu-based raytracing, 2023.

[14] Greg Smolka. How Far Down the Road Is the Autonomous Vehicle? https://www.photonics.com/Articles/How_Far_Down_the_Road_Is_the_Autonomous_Vehicle/a66856-power-point-talha/. [Accessed 17-01-2024].

[15] Josef Spjut, Solomon Boulos, Daniel Kopta, Erik Brunvand, and Spencer Kellis. Trax: A multi-threaded architecture for real-time ray tracing. pages 108 – 114, 07 2008.

[16] Kate Yurkova. A Comprehensive Overview of Gaussian Splatting — medium.com. https://medium.com/towards-data-science/a-comprehensive-overview-of-gaussian-splatting-e7d570081362. [Accessed 13-01-2024].