Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

Integració a MATLAB de la visualització volumètrica mitjançant l'algorisme de Ray Casting basat en GPU

Ernest Albets Moreno

Directors: Anna Puig, Sergio Escalera i Frederic Sampedro
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 20 de juny de 2013

Índex de la memòria:

Índex de figures i taules:

Abstract

Nuclear medicine has been a great advance for medicine in general. Thanks to this discipline is possible accurately detect pathological areas in different parts of the body using devices such as scanners,RMI-PETs, SPECT or any other device that uses radiation. This become a great benefit for diagnose, detection and disease prevention.

To facilitate the task of nuclear physicians, in this project a software application will be developed to able to manage RMI-PET medical volumes and to deal with a graphic interface that renders the volume with distinct pathological areas. Moreover, it will allow to interact with the volume modifying several parameters such as the view position, zoom areas, contrast parameters... etc, in order to enhance the professionals' experience where they will be able to classify the patient's condition based on the results.

One of the biggest problems in MATLAB are the slow speed in the operations with hugh matrices. Consequently, the display of volumes is slow and tedious. To speed and to solve this problem, it is necessary an external rendering algorithm capable to increase image processing in MATLAB.

RayCasting's algorithm is one of the best algorithms for image processing due to its quality. Moreover, the use of the paralelism that offers recent GPUs allows to process graphics speedier than CPU would do.

Along this project we will see how to integrate in this application an external Ray Casting algorithm based on GPU which will be capable of improving the problem of slow display. Furthermore, it will be analyzed and discussed in detail with real datasets and we present the obtained results, improvements and developed implementation.

1. Introducció

L'objectiu principal del projecte es centra en accelerar la velocitat de visualització (renderització) de volums mèdics capturats a partir de ressonància magnètica o escàners, per tal de poder realitzar posterorment una identificació de zones amb alta activitat que podrien esdevenir tumors i poder efectuar una classificació.

Per tal de millorar la visualització dels volums, s'utilitzarà l'algorisme de Ray Casting de volum basat en el mode MIP (*Maximum Projection Intensity*) i implementat sobre la GPU. El resultat del projecte s'integrarà a una aplicació prèviament implementada amb MATLAB que conté la interfície gràfica necessària per interactuar, així com diferents volums mèdics RMI-PET (ressonància magnètica- imatges funcionals) reals amb les seves màscares de segmentació .

1.1 Àmbit del projecte

En l'anàlisi de les dades mèdiques per especialistes per realitzar diagnòstics i planificar operacions, els metges especialistes en medicina nuclear utilitzen eines de visualització i de detecció de les zones o estructures patològiques.

Per tal que es puguin visualitzar de forma interactiva diferents regions o estructures anatòmiques de dades captades directament des d'aparells com ressonàncies magnètiques, escàners i imatges funcionals, com PET i/SPECT, és necessari realitzar un procés d'etiquetatge o classificació. Els metges en medicina nuclear analitzen primer les imatges d'activitat funcional per identificar les estructures patològiques i visualitzen aquestes zones en les imatges procedents d'escàners per situar les patologies en l'anatomia del pacient.

En aquest projecte es pretén desenvolupar tota la interfície d'interacció amb algorismes de classificació ja desenvolupats prèviament en MATLAB i connectar un visualitzador extern basat en Ray Casting de volum sobre la GPU per accelerar la visualització de **MATLAB[11]**.

El projecte és integrable a una aplicació ja implementada en MATLAB que carrega volums mèdics obtinguts per ressonància magnètica i que a partir de l'aplicació d'una màscara de segmentació a cada volum, determina les zones patològiques que poden contenir un tumor segons l'activitat resultant. El resultat d'aquesta és força lent i no dóna un resultat 100% fidedigne, ja que zones que usualment ja tenen una alta activitat (cor, estómac...) serien detectades com a patològiques (falsos positius). Per aquest motiu, el resultat de la màscara de segmentació de l'aplicació descarta aquestes zones i es centra en aquelles que tenen un resultat que no s'ajusta a la realitat (zones usualment amb baixa activitat que són detectades com a patològiques per tenir massa activitat).Dita aplicació, conté també implementada una interfície gràfica senzilla, simple i intuïtiva, que permet la interacció amb l'usuari on, entre altres coses, podrà carregar el seu volum, posar-hi la màscara corresponent, i podrà anar movent el volum mèdic obtingut a clic de ratolí. A partir dels resultats obtinguts, tenint en compte el que s'ha explicat, l'especialista podrà adjudicar un diagnòstic al pacient a la vegada que podrà classificar l'estat d'aquest.

L'aplicació esmentada, i per tant, el projecte present, tindrien la seva funcionalitat final en l'Hospital Sant Pau de Barcelona. Per tant, el perfil d'usuari que l'utilitzaria, seria, en principi, un especialista mèdic en oncologia. Aquest fet, requereix que els resultat siguin el més fiables possibles i que no tinguin un alt cost computacional ni en temps ni en memòria.

Finalment, relacionant-ho amb el pla d'estudis de la carrera, han estat útils moltes de les assignatures impartides tant en el Grau d'Enginyeria Informàtica (cursat del 2011 al 2013) com l'antiga ETIS (Enginyeria Tècnica en Informàtica de Sistemes), cursada del 2008 al 2011, per a la implementació, realització i disseny d'aquest projecte. La que ha estat de més utilitat, i sens dubte la base sobre tot el que ha envoltat a aquest projecte, ha sigut la de Gràfics i Visualització de dades, que donava les eines i frameworks necessaris per entendre i introduir-nos en el món de les aplicacions gràfiques, programar sobre la GPU, tenir nocions sobre els elements que hi pot haver en una escena i saber-los calcular, així com visualitzar i obtenir imatges en un widget.

En menor proporció, també han estat útils les assignatures següents:
- Elements de Programació, per ser la introductora en tots els temes referents a programació i llenguatges d'alt nivell sense la qual no es seria capaç de realitzar cap aplicació software.

- Estructura de dades, que amplia el coneixement sobre les possibles estructures d'emmagatzematge de dades possibles així com el cost temporal i computacional de cada una d'elles.

- Metodologia i tecnologia de la programació, que ensenya conceptes tan útils con l'abstracció i permet realitzar els dissenys, diagrames i plantejaments previs abans de tota implementació.

- Visió Artificial, per donar nocions sobre visió per computador, definir conceptes tan essencials com píxels, màscares o classificadors i per endinsar-nos en la programació amb MATLAB.

- Sistemes Operatius I i II, on aprenem a programar amb el llenguatge C i a saber com el nostre computador gestiona i planifica els seus recursos interns. També ens introdueix en el món de la programació concurrent.

1.2 Motivació

Havent introduït en l'apartat anterior l'àmbit del present projecte, s'ha mencionat que un dels principals problemes que presentava l'aplicació mèdica era la lentitud amb la que visualitzava i processava els volums mèdics proporcionats. Es tracta doncs, d'un efecte molt perjudicial si tenim en compte la utilitat de la mateixa, pel que la principal motivació ha estat centrar-se en accelerar la velocitat de renderitzat dels volums mèdics segmentats i obtenir així un temps de visualització que s'adeqüi a les exigències requerides (disminuir el cost temporal).

Per donar solució al problema, s'ha optat en utilitzar l'algorisme de processament d'imatges "Ray Castig" (traçat de rajos), i en aquest cas particular, el Ray Casting de volum, ja que es disposa de volums mèdics tridimensionals.

Aquest algorisme és un dels més utilitzats per processar imatges donada la seva qualitat final en les visualitzacions i la seva potencialitat de ser implementat a la GPU. Bàsicament, es basa en la creació de rajos des del punt de visió fins a cada píxel on és mira la intersecció amb l'objecte i altres característiques d'il·luminació per determinar les superfícies visibles de l'objecte. El **Ray Casting de volum[20]** pot efectuar-se tant en CPU com en GPU, però es va determinar fer-lo sobre la GPU ja que la velocitat és molt més bona degut a l'ús del paral·lelisme. Altrament, dins d'aquest hi ha diferents modes de visualització. Per a les característiques de la nostra necessitat, el millor era utilitzar el mode **MIP[19]** (Maximum Intensity Projection), que es queda amb el valor màxim obtingut en la intersecció.

Els llenguatges utilitzats són el **C++[2]** i el **GLSL[4]** , que són els que es van utilitzar a l'assignatura de Gràfics i Visualització, ideals per a programar sobre la GPU.

1.3 Objectiu general

L'objectiu principal del projecte consisteix en l'anàlisi, disseny i implementació del mètode de Ray Casting de volum en mode de projecció de màxima intensitat (MIP) sobre la GPU que permeti la connexió i integració amb MATLAB mitjançant una llibreria externa.

1.4 Objectius específics

L'objectiu general anterior es pot concretar en els següents subobjectius:

1. Anàlisi, disseny i implementació del mètode de Ray Casting sobre volums en mode MIP:

   1.1 *Desenvolupament del mètode de Ray Casting de volum mode MIP en la GPU*

   1.2. *Integració de la màscara del subvolum d'interès a emfatitzar en el mètode de Raycasting.*

2. Anàlisi, disseny i implementació d'una interfície prototipus que permeti interaccionar amb el mètode de Ray Casting des de la plataforma de MATLAB:

   1.1 *Definició d'un model de càmera.*

   1.2 *Definició de diferents paràmetres: angles de visió, contrast.. etc, que permeten el canvi de paràmetres de visualització.*

   1.3 *Definició dels models de vòxels i de regions patològiques a analitzar.*

3. Anàlisi dels resultats amb diferents conjunts de dades reals, avaluant la qualitat de les visualitzacions i el temps de procés de la visualització

1.5 Organització de la memòria

La present memòria s'estructura amb els capítols següents:

- **Capítol 1: Introducció**

  Breu explicació introductòria sobre la temàtica del projecte. Situa i relaciona el projecte dins de la seva temàtica general i amb el pla d'estudis del Grau d'Enginyeria Informàtica. S'explica també l'origen del projecte i la seva necessitat de dur-se a terme. Finalment descriu de forma concisa l'objectiu general del projecte i se'n desglossen els subobjectius.

- **Capítol 2: Antecedents**

  Anàlisi dels antecedents del tema. Permet situar el projecte dins un context més general. S'explica el renderitzat en 3D i altres termes importants, la funcionalitat de la segmentació i classificació volumètrica i la necessitat d'interacció amb volums. Finalment extreu conclusions dels apartats previs.

- **Capítol 3: Anàlisi**

  Mostra i explica els diferents casos d'ús i el model de domini de l'aplicació.

- **Capítol 4: Disseny**

  Mostra i explica el diagrama de classes i els diagrames de seqüencia associats a cada cas d'ús.

- **Capítol 5: Implementació**

  Argumenta i detalla el desenvolupament intern del projecte i els termes referents a la implementació.

- **Capítol 6: Resultats**

  Il·lustra i comenta les simulacions fetes: descripció i visualització dels resultats obtinguts de l'aplicació amb captures de la interfície. Conté també una taula comparativa dels costos temporals calculats amb els diferents algorismes i

paràmetres d'aquests.

- **Capítol 7: Conclusió**

  Conclusions extretes de l'objectiu del projecte i llistat de possibles millores o línies obertes futures.

- **Capítol 8: Referències bibliogràfiques**

- **Apèndix A: Manual tècnic**

  A.1 Instal·lació, requeriments mínims i passos a seguir.

  A.2 Manual del desenvolupador

- **Apèndix B: Manual de l'usuari de l'aplicació.**

2. Antecedents

En el present capítol es realitzarà un anàlisi exhaustiu sobre els antecedents del tema tractat pel projecte per tal de situar-lo millor en un context més genèric i informar sobre aspectes teòrics rellevants que ajudaran a entendre la temàtica i el funcionament dels algorismes i elements del projecte . Es desglossa en el següents cinc subapartats:

2.1 *Visualització de volums*

2.2 *Classificació de volums*

2.3 *Interacció amb volums*

2.4 *Aplicació de partida en MATLAB*

2.5 *Conclusions*

2.1 Visualització de volums

Al llarg d'aquest subcapítol s'exposaran i detallaran processos i elements importants a l'hora de visualitzar volums tals com: volum, vòxel, món de vòxels, renderització 3D, càmera, escena, Ray Tracing, Ray Casting i Ray Casting de volum (en la GPU i en mode MIP).

2.1.1 Definició de volum, vòxel i món de vòxels

Un dels avantatges més importants de la disciplina de gràfics per computador és la possibilitat de processar un conjunt de dades tridimensionals i poder visualitzar-les i projectar-les en una escena bidimensional. Aquest fet de poder generar una imatge bidimensional a partir d'una escena tridimensional rep el nom de **renderització[16]** (o visualització), i és la base del processament d'imatges 3D ***(veure Fig.1)***.

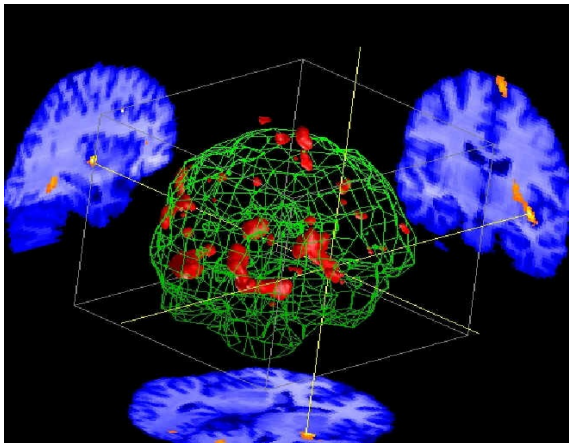


Fig.1- Renderització volum mèdic RMI

http://psyphz.psych.wisc.edu/~oakes/spam/spam_BrainSpinner.htm

Un **volum** mèdic extret per la tècnica RMI-PET o per escàners, no és res més que una estructura tridimensional de dades que s'organitzen d'una manera determinada. Cada una de les unitats del conjunt de dades rep el nom de vòxel.

Un **vòxel[21]** és com un píxel però en tres dimensions. Així, si un **píxel** representa una densitat o nivell de gris en una matriu bidimensional (unitat mínima informativa en una imatge 2D), un vòxel és la representació d'un valor de l'objecte o matriu tridimensional. Per tant, un vòxel no és mai un volum sinó un punt del conjunt de dades del volum i la seva ubicació pot determinar-se mitjançant tres coordenades.

Les propietats que pot representar un vòxel són diverses, però pot determinar des d'una opacitat fins a un color i una opacitat.

Si un vòxel és només una partícula que conforma el conjunt de dades d'un volum mèdic en un espai de tres dimensions, tot el conjunt de vòxels que forma un volum s'anomena **món de vòxels[22]** ***(veure Fig.2)***.

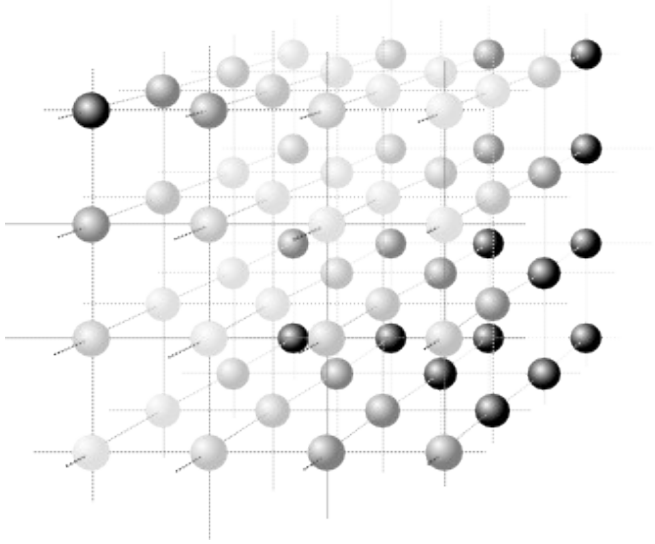


Fig.2 - Món de vòxels en l'espai tridimensional.

http://commons.wikimedia.org/wiki/File:Voxelgitter.png

Un **món de vòxels** pot esdevenir una quantitat de dades immensa que necessita organitzar-se i estructurar-se correctament. A vegades resulta difícil processar-lo degut a les limitacions de molts ordinadors en espai de memòria o en ample de banda. Generalment, els vòxels que componen el món de vòxels determinen la mida del volum i es reparteixen per cadascun dels eixos d'aquest. Això, i la mida que ocupi un vòxel particular determinaran la quantitat d'espai a ocupar pel volum.

2.1.2 Renderització 3D:

Per tal de poder processar i visualitzar les estructures de dades tridimensionals descrites anteriorment, haurem de sotmetre el món de vòxels dels nostres volums mèdics RMI-PET per un procés de **renderització 3D[16].**

La renderització és el procés de generació d'una imatge 2D o vídeo que calcula la il·luminació global a partir d'un model 3D donat. En termes generals, seria la projecció 2D, en una escena, d'un model de tres dimensions, com pot ser perfectament el món de vòxels que forma un volum mèdic RMI-PET.

El terme renderització no designa a una tècnica en concret sinó a un conjunt de tècniques diferents que poden aplicar-se per visualitzar models 3D. Algunes de les tècniques més emprades són el **Z-Buffer[24]**, el **Shear-Warp[1]**, el **Ray Tracing[15]** o l'algorisme en què es basa aquesta última, el **Ray Casting[15]**, on només es considera el raig primari i és el que s'utilitza en el present projecte adaptat per processar els volums.

Per poder dur a terme el procés de renderitzat en un volum és necessària la determinació d'un color i una opacitat per cada valor de la densitat del vòxel d'aquest. Per a la determinació del color, es necessiten tres canals: RGB (Red Green Blue), on R indicarà la intensitat de color en el canal vermell, G en la de verd i B en la de blau ***(veure Fig.3)***. L'opacitat vindrà determinada per un canal més, l'anomenat A (Alpha), que conjuntament amb el altres tres, donarà lloc al RGBA final. És una funció de transferència ***(veure Fig.4)*** la que realitza el pas de densitat d'un vòxel a un RGBA concret, que, finalment, es projectarà als píxels del framebuffer donant el resultat en la imatge 2D d'un widget. Segons la tècnica utilitzada el procediment i els resultats seran d'una manera o una altra.

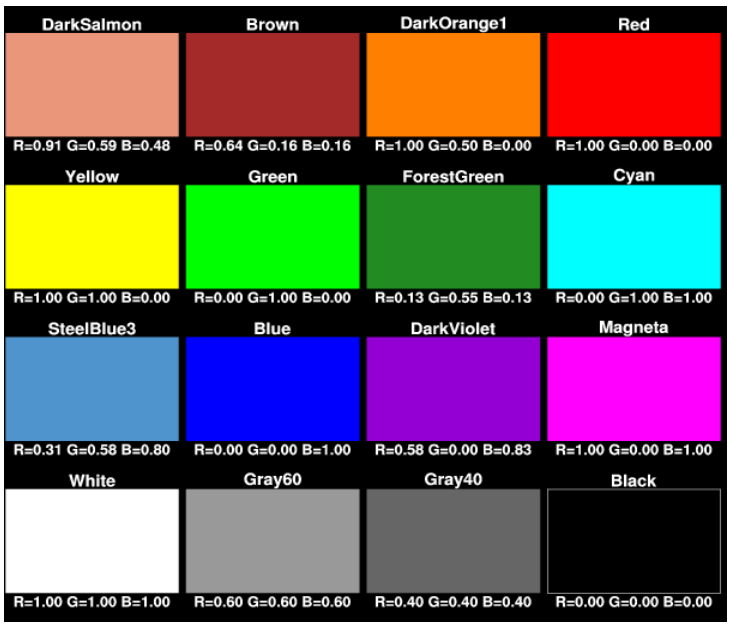


Fig.3 – Taula de 16 colors bàsics amb la seva combinació RGB

http://www.pyngl.ucar.edu/Graphics/create_color_table.shtml

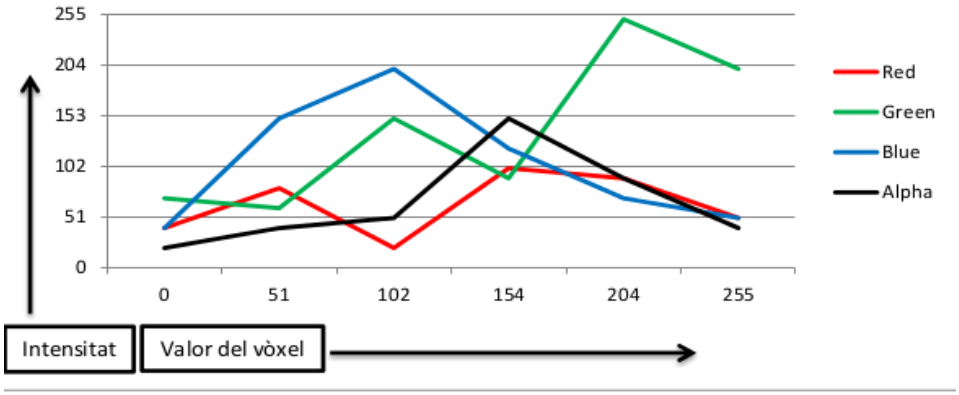


Fig.4 – Funció de transferència entre valors i colors (Look up Table)

2.1.3 Definició model càmera i escena:

Per aconseguir que el nostre volum a renderitzar pugui veure's correctament en un widget, necessitarem la definició d'una **escena**, una **càmera** que permetrà que aquest pugui veure's des de diferents punts de vista i amb diferents tipus de projeccions, i, tot i no ser imprescindibles, també es pot necessitar una o diverses **llums** per aplicar mètodes d'il·luminació que detallin l'escena i els objectes d'aquesta, o l'aplicació de **materials** i

textures per detallar els objectes i donar sensació de versemblança. En el present projecte, no ha estat necessària la introducció de llums i materials, ja que es considera que cada vòxel emet i absorbeix una quantitat de llum segons el valor de densitat que té associat. Aquests valors es defineixen a partir de la funció de transferència abans esmentada.

Una **escena** es defineix com una capsa contenidora d'objectes a visualitzar. Es necessita calcular la capsa mínima contenidora dels objectes per determinar l'espai de l'escena que es visualitzarà sobre el conjunt d'objectes que aquesta conté.

La **càmera[6]** és l'element imprescindible per generar la visualització final dels objectes definits. És la responsable de que els objectes es vegin d'una manera determinada: més propers o més llunyans, sencers o retallats, en diferents vistes, perspectives o angles de visió diferents. Tècnicament, seria el punt de vista des d'on s'observa l'escena i el lloc on es situa l'observador. El resultat de la càmera sobre l'escena es visualitzarà en el **viewport** definit. El viewport és un rectangle 2D on es projecta l'escena 3D a la posició de la càmera, és una regió de la pantalla utilitzada per mostrar total o parcialment la imatge a ser mostrada. El viewport s'emmarcarà en un widget que serà qui controlarà les interaccions de l'usuari en la visualització.

Una càmera conté uns determinats **angles de visió** que determinaran la vista o posició de l'objecte o objectes a visualitzar. A part, per realitzar els seus càlculs, també és necessari determinar el **VRP** (View Reference Point) que és el punt cap on mira l'observador, el **VUP** (view-UP vector) que és un vector que indica la verticalitat de la càmera i una **distància "d"** entre l'observador i el VRP. Altres elements importants són el **Frustum** i l'**Aspect Ratio**. El Frustum és un tros de con que delimita l'espai visible de l'escena i ve delimitat pels plans de retallat (clipping): pla de retallat anterior, pla de retallat posterior i plans de retallat laterals (depenen de la projecció). L'Aspect Ratio és la relació entre l'amplada i alçada de la finestra de projecció 3D de l'escena i del viewport final de la renderització ***(veure Fig.5)***.

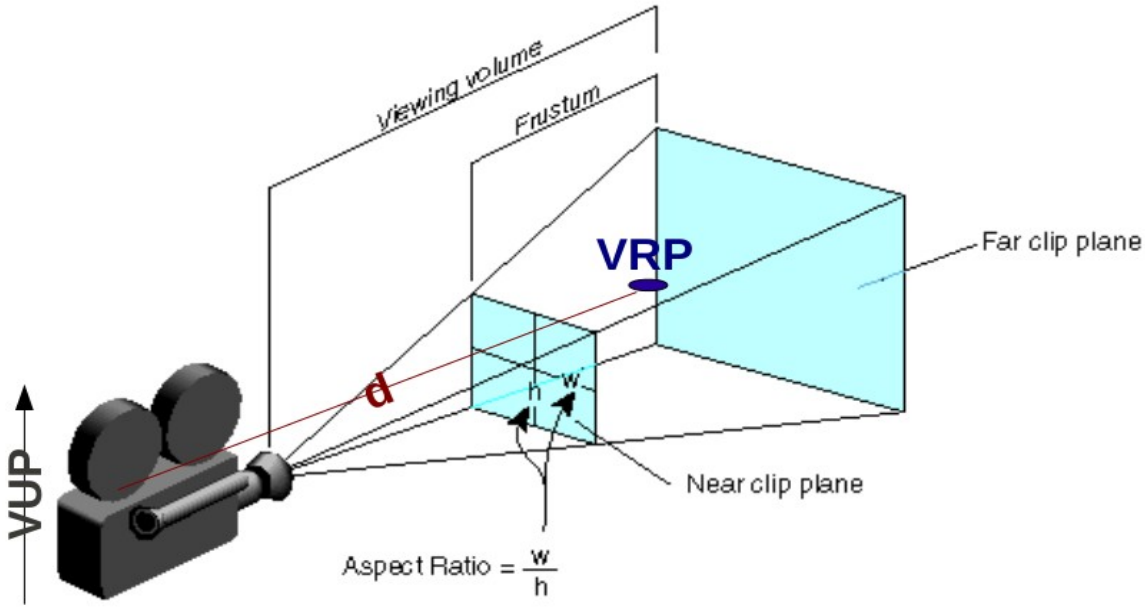


Fig.5- Model Càmera [6]

És possible que una càmera estigui definida en **clipping (retallat)**. D'aquesta manera, només podrà veure's una part del món o l'escena i els objectes que quedin fora del Frustum seran retallats. A més a més, la projecció al pla pot ser de diferents tipus: una projecció planar **perspectiva**, on les línies de projecció convergeixen cap al centre de projecció o una projecció planar **paral·lela** on les línies de projecció són paral·leles ***(veure Fig.6)***.

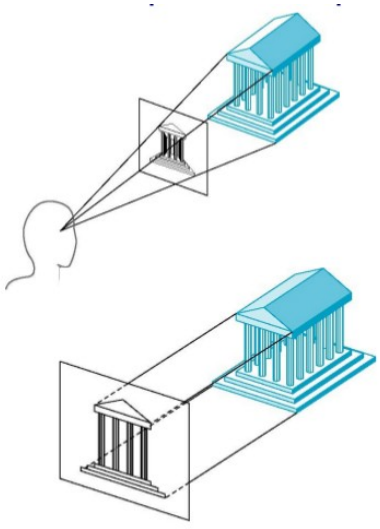


Fig-6-Projecció perspectiva (superior) i Projecció paral·lela (inferior) [6]

Finalment, per passar els objectes de l'escena del sistema de coordenades del món al sistema de coordenades de la càmera es requereix d'una sèrie de transformacions afins realitzades per matrius de 4x4. La matriu de **model-view[7]** és l'encarregada de fer el canvi de coordenades del món a coordenades de la càmera . Amb el càlcul d'aquesta matriu, la **matriu de projecció[7]** i les transformacions necessàries (rotació, translació...) es pot obtenir el resultat final.

2.1.4 Tècniques de renderització: Ray Tracing, Ray Casting i Ray Casting de volum

Una de les tècniques de renderització 3D més utilitzades degut als bons resultats en qualitat d'imatge és l'anomenada “Ray Tracing” o traçat de rajos. Es tracta d'una tècnica complexa però que dóna uns resultats de molt alta qualitat i a una velocitat acceptable, superant la tècnica del Z-Buffer (que no dóna imatges molt realistes) i la tècnica de Radiosity que és molt lenta.

La tècnica del **Ray Tracing[15]** consisteix en traçar una sèrie de rajos des de l'observador (càmera) contra l'escena (tants rajos com píxels). Posteriorment es van calculant les interseccions dels rajos amb els objectes, i aquests, van rebotant de forma recursiva per calcular l'aportació de la llum a altres objectes ***(veure Fig.7)***. Per tant, a partir de la intersecció del primer raig es calcula el segon raig reflectit en aquell punt d'intersecció que buscarà la il·luminació del primer punt intersecant en una altra superfície. D'aquesta manera, es poden modelar objectes transparents o miralls amb múltiples reflexions.

A l'hora de traçar el rajos des de l'observador les superfícies visibles són determinades a partir d'un procés d'ombrejat (intensitat del píxel) i es tenen en compte factors d'il·luminació com reflexions, refraccions o ombres.

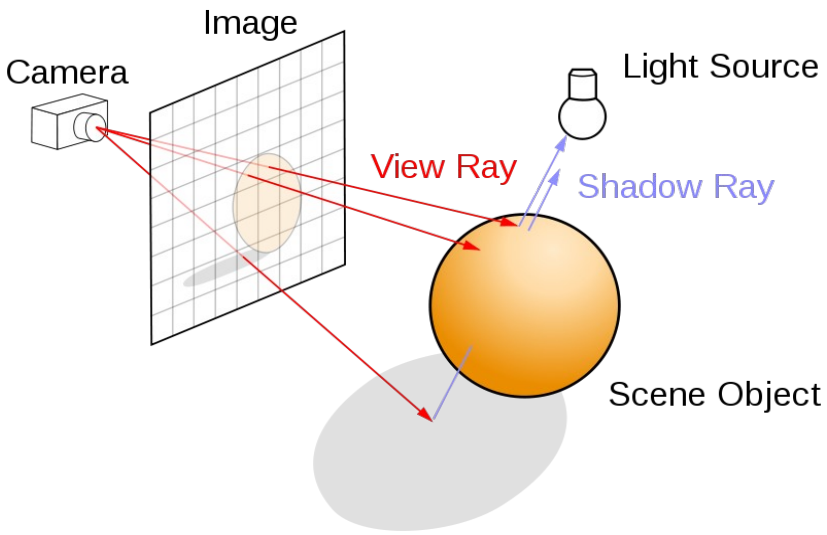


Fig.7 - Exemple de Ray Tracing

http://www.codinghorror.com/blog/2008/03/real-time-raytracing.html

Un cas particular de l'algorisme de Ray Tracing és l'algorisme de determinació de superfícies visibles anomenat **Ray Casting[14]**. El **Ray Casting** utilitza interseccions de raig-superfície per obtenir solucions a diferents problemes i només té en compte el primer raig d'intersecció amb l'escena, sense utilitzar la recursivitat de l'algorisme de Ray Tracing. No s'ha de confondre, doncs, Ray Tracing amb Ray Casting, ja que el primer és un algorisme que es basa en el que realitza el segon. Tampoc cal confondre Ray Casting amb **Ray Casting de volum.** Aquest últim, és el que utilitzem en el projecte present i processa dades tridimensionals. Per contra, el Ray Casting només processa dades de superfície. En el **Ray Casting** es realitza el mateix procés de traçat de rajos des de l'observador fins la superfície i dóna com a objecte visible aquell la intersecció raig-objecte del qual sigui la més propera a l'observador.

El que es tracta en aquest projecte són volums mèdics RMI-PET formats per móns de vòxels. En conseqüència, s'ha de realitzar l'algorisme Ray Casting adaptat per processar dades tridimensionals amb l'anomenat **Ray Casting de volum**. El **Ray Casting de volum[20]** computa imatges 2D a partir de conjunts de dades tridimensionals de volums. Ara, per a cada píxel de la imatge final que volem obtenir es llança un raig cap el volum des de la posició de l'observador. Cada raig passarà a través d'una sèrie de vòxels del volum. El que es tracta és doncs d'assignar una única opacitat i color (RGBA) per cada raig traçat del conjunt de vòxels per on passa i associar-ho al píxel de la imatge final ***(veure Fig.8)***.

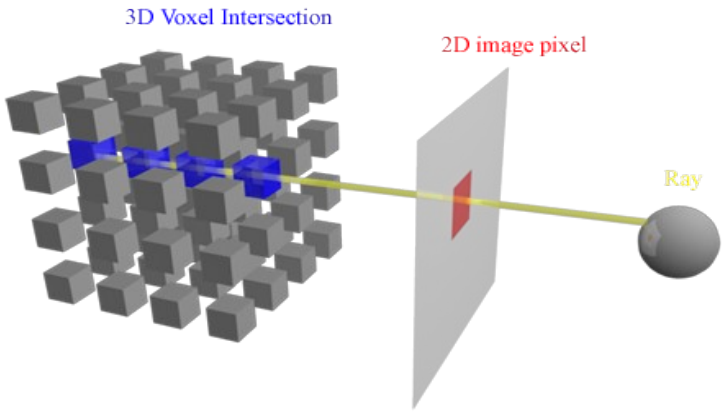


Fig.8 -Ray casting de volum, raig d'un píxel travessant una sèrie de vòxels

http://www.volviz.com/topics.php

Els resultats finals són molt realistes i de gran qualitat segons el tipus d'il·luminació que s'utilitzi ***(veure Fig.9)***.

Fig.9 -Imatge processada utilitzant Ray Casting de Volum

http://en.wikipedia.org/wiki/File:High_Definition_Volume_Rendering.JPG

2.1.5 Ray Casting de volum en mode MIP

Cal destacar que el Ray Casting de volum pot implementar-se en diferents modes. Si bé la mecànica de l'algoritme és l'exposada en l'apartat 2.1.4, existeixen diferents tècniques per obtenir el color i opacitat final del píxel després d'haver llançat el raig que travessa una sèrie de vòxels.

El mode **MIP (Maxiumum Projection Intensity)[9]** és un mètode senzill i simple que el que realitza és bàsicament quedar-se amb la intensitat o valor màxim de tots els valors contigunts en el conjunt de vòxels travessats pel mateix raig. Aquest valor màxim és el que després serà pintat en el píxel corresponent en forma de color RGBA. Els renderitzats que s'obtenen són similars a les radiografies fetes amb Rajos X ***(veure Fig.10)***.

Fig.10 - Ray Casting de volum utilitzant mode MIP

Els avantatges del mode MIP són, a part de la seva senzillesa, el fet de no necessitar cap funció de transferència per obtenir bons resultats (no com, per exemple, en el mode de DVR *Direct View Rendering*), fet pel qual resulta ideal per processar imatges provinents d'escàners o de ressonància magnètica. Tot i així, també té una sèrie d'inconvenients que és necessari remarcar. El principal d'ells, és la pèrdua del context espacial degut a la independència d'ordre espacial (proximitat o llunyania) que hi ha a l'hora de quedar-se el màxim valor, per tant, hi ha una manca d'informació de la profunditat que pot generar certes ambigüitats en la imatge final mostrada. Una possible solució a aquest problema seria realitzar una atenuació en profunditat, emfatitzar les siluetes i combinar el mètode DVR (que sí conté informació de la profunditat, però conté una funció de transferència complexa), amb el mètode MIP, donant lloc a un mètode anomenat **MIDA[9]** (*Maximum Intensity Difference Accumulation*).

2.1.6 Ray Casting de volum a la GPU

Un cop exposat el funcionament de l'algorisme Ray Casting de Volum, cal especificar que aquest pot ésser implementat tan en la **GPU** com en la **CPU**.

El problema de la CPU (*Central Processment Unit*) és la seva lentitud i les seves limitacions tant temporals com espacials (capacitat) a l'hora d'emmagatzemar, processar i treballar amb grans conjunts de dades en temps real, degut al gran treball al que està sotmesa. Si tenim en compte que els volums mèdics estan formats per un gran conjunt de dades (els vòxels), necessitarem una gran capacitat de memòria i emmagatzematge, sense limitacions en les dimensions. A més a més, la interactivitat denota treballar veloçment en temps real. En definitiva, si el nostre interès és optimitzar la velocitat de renderització el màxim possible, l'opció de la CPU quedaria descartada, donant lloc a una implementació sobre la GPU.

La **GPU (*Graphic Processment Unit*)** ha esdevingut en els darrers temps un element de gran importància que ha marcat un punt d'inflexió pel processament d'imatges per computador, aplicacions gràfiques i totes les disciplines que comprenen els gràfics per computador. Aquesta unitat ha fet que es pugui treballar d'acord amb les demandes i exigències actuals (cada vegada majors) en sectors com el de videojocs o la medicina nuclear. El fet de contenir una sèrie elevada de processadors interns que treballen en paral·lel, li confereixen una característica molt important que és el paral·lelisme. Si a això i sumem que tenen una potència de càlcul millor que la CPU, i que és complexa i altament programable SIMD (*Single Instruction Multiple Data*), les unitats de processament gràfic reuneixen totes les característiques necessàries per convertir-se en les millors a l'hora de processar elements que requereixen de visualització gràfica ***(veure Fig.11)***.

Queda patent doncs, que un **Ray Casting de volum basat en GPU[5]** per volums mèdics RMI-PET, donarà uns resultats òptims, de gran qualitat, sense problemes en espai o memòria, amb major flexibilitat i velocitat d'interacció en temps real.

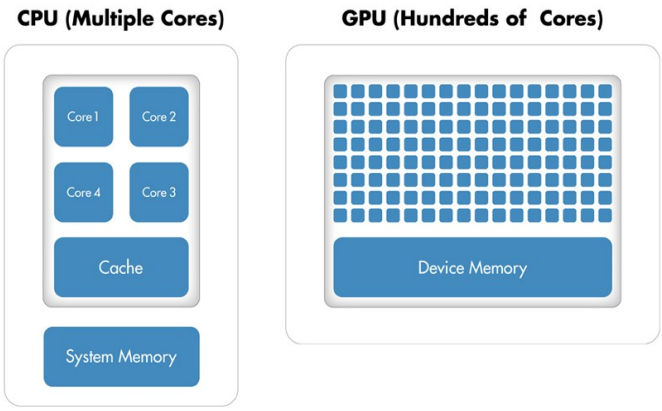


Fig.11 -Esquema CPU vs GPU

http://www.mathworks.es/company/newsletters/articles/gpu-programming-in-matlab.html

Per a poder treballar i escriure aplicacions que produeixin gràfics tant en 2D com en 3D existeix una especificació estàndard que defineix una API multillenguatge denominada **OpenGL[12]**. Aquesta, conté una sèrie de funcions, llibreries i frameworks que ens permeten implementar aplicacions gràfiques. Per a programar aplicacions sobre la GPU existeixen uns frameworks basats en OpenGL denominats "shaders". Els **shaders** permeten una interacció directa amb la GPU, es compilen de forma independent i utilitzen un llenguatge d'ombrejat, com el **GLSL[4]**, utilitzat en aquest projecte. Dos dels tipus de shaders utilitzats són el **fragment shader** i el **vertex shader**. El primer d'ells, treballa en paral·lel a nivell de píxels i aprofitant les seves propietats de looping, és on s'implementarà l'algorisme de Ray Casting de Volum. L'altre treballa a nivell de vèrtexs els quals pot transformar (coordenades, color...) i les seves sortides seran entrades del fragment shader.

2.1.6.1 Ray Casting de volum a la GPU: Tècnica *One Step*

L'algorisme de Ray Casting de volum sobre la GPU pot efectuar-se en dos passos de renderitzat, o pel contrari, utilitzar una tècnica que optimitzaria aquest procés reduint els dos passos en un: ***One Step***. Aquest últim és l'efectuat en aquest projecte ja que el seu cost temporal és significativament inferior a l'altre i per tant, resol més bé la necessitat d'accelerar el procés de renderitzat en l'aplicació dels voulms.

Si tenim en compte que per dur a terme la tècnica de Ray Casting de volum cal enviar un raig per a cada píxel de la imatge final i mostrejar el color corresponent al vòxel triat dels travessats pel raig (independentment del mètode efectuat), s'usarà el paral.lelisme de la GPU per processar simultàniament cadascun dels raigs.

OpenGL facilita l'activació d'un procés per cada píxel on es projecta una determinada geometria. Es pot fer ús d'aquesta propietat visualitzant el cub contenidor del volum a renderitzar per activar un procés per cada píxel projectat. A cadascun dels píxels es genera un raig.

Cadascun dels raigs és una línia recta formada per una equació que té el punt d'origen en coordenades del món, un increment de desplaçament i un vector director *(pOrigen + increment*vector director)*. Per a obtenir aquesta informació s'utilitza el cub projectat.

Es renderitza el cub de la mateixa mida que el món de vòxels del volum a processar i , com a color de cadascun dels vèrtexs es defineixen les coordenades del punt. En OpenGL, una geometria es pot renderitzar de dues maneres: via *frontface* (cares frontals) o via *backface* (cares posteriors) ***(veure Fig. 12)***. Per obtenir el punt d'origen d'un raig corresponent a un cert píxel, s'agafarà el punt actual del cub en el píxel corresponent de la imatge generada en mode *frontface*. En el procés dels dos passos de renderitzat, el vector director es troba com a: "*backface – frontface*". Això fa que per calcular el raig calgui renderitzar tant les *frontface* del cub com les *backface*, desembocant en un cost temporal i computacional significativament costosos, ja que caldrà inicialitzar dos framebuffers i dos renderbuffers i fer passos de procés en la CPU.

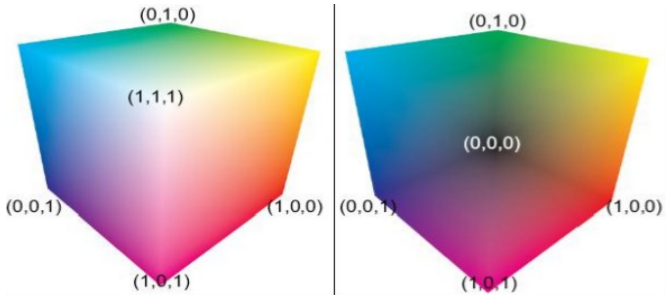


Fig. 12 – Cub renderitzat en *frontface*'(esquerra) i en *backface* (dreta)

Si aprofitem la informació i les dades que conté el model de la càmera, podem reduir els dos passos en un sol pas (*One Step*) i així evitar-nos tots els desavantatges que comporta el renderitzat en dos passos i optimitzar tot el procés del renderitzat de Ray Casting de volum en la GPU. El pas que es pot estalviar és el del renderitzat de les *backface*, evitant passos a la CPU i executant abans els shaders de la GPU.

Es tracta doncs de calcular el vector director d'una forma alternativa a la resta entre les *frontfaces* i les *backfaces*. La càmera conté dos paràmetres que permeten realitzar el càlcul del vector director sense necessitat d'acudir al les *backfaces* del cub: el VRP (posició d'origen del volum, *descrit en el subcapítol 2.1.3*) i l'observador (posició on es troba la càmera, *descrit en el subcapítol 2.1.3*). Utilitzant aquestes dades podem obtenir el vector director del raig com a: "*VRP – Observador*", permetent renderitzar en la GPU en un sol pas.

Per tant, caldrà passar totes las dades necessàries cap als shaders de la GPU per poder dur a terme el Ray Casting de volum sobre aquesta: coordenades de les textures del cub per renderitzar les *frontface*, punts del cub, textura del volum i textura de la màscara per representar el món de vòxels i la segmentació...etc. Finalment, en el fragment shader, aprofitant el seu treball paral·lel a nivell de píxels, es durà a terme l'algorisme del Ray Casting de volum on es calcularà el raig i es mostrejaran el colors dels píxels resultants aplicant el mode MIP.

2.2 Classificació de volums

Un cop introduït el procés de visualització dels volums mèdics RMI-PET, és necessari, i especialment en el context de la medicina nuclear, argumentar la necessitat d'un procés de classificació d'aquests a partir de segmentacions que ajudin a tal propòsit.

Una **segmentació[8]** d'un volum 3D consisteix en efectuar una divisió de zones o àrees específiques de vòxels d'aquest que continguin una informació rellevant o significativa respecte la resta. Aquests regions segmentades han de ser perfectament distingibles de la resta, pel que una bona opció és pintar en un color diferent aquestes zones determinades ***(veure Fig.13)***. Algunes tècniques de segmentació conegudes són: el **thresholding[18]**, que estableix un llindar i agrupa els vòxels amb un valor superior a aquest llindar en un grup i la resta en un altre, i el **clustering[3]**, que agrupa vòxels en diferents classes que tinguin característiques similars.

La finalitat dels volums mèdics obtinguts per RMI-PET o escàners en la medicina nuclear és la de detecció de zones patològiques per tal de poder establir una sèrie de diagnòstics sobre l'estat dels pacients i una possible prevenció associada a aquest. En el present projecte es duu a terme una segmentació dels volums processats que diferencia explícitament aquestes zones patològiques de la resta.

Tal i com ja s'ha explicat, es determina com a zona patològica a aquella que ha donat una alta activitat en passar-hi la radiació. Hi ha zones que per defecte ja disposen d'una elevada activitat degut a la funció dels seus òrgans interns (zones com les del cor, l'aparell reproductor o en alguns casos l'estómac). En canvi, la resta de zones que contenen òrgans poc actius, haurien d'aparèixer sempre amb baixa activitat. La dificultat i limitació d'aquesta classificació radica en la detecció d'una zona patològica continguda en una zona d'alta activitat per defecte, ja que en ser difícil determinar si es tractarà d'una zona patològica o una elevada activitat normal, es descarta com a zona patològica en la segmentació final. Per altra banda, sí que es determina una zona com a patològica si aquesta es troba en un sector on hi hauria d'haver baixa activitat.

En aquest projecte, cada volum mèdic RMI-PET conté la seva pròpia màscara de segmentació binària (amb zeros i uns). Aquestes màscares han estat realitzades prèviament per segmentació manual o automàtica. La utilitat d'aquesta màscara és la d'establir i diferenciar les zones hipotèticament patològiques de la resta. Així, els zeros representarien zones no patològiques, mentre els uns detectarien les zones patològiques.

Aplicant aquesta màscara binària al seu volum corresponent es veurà el resultat final del volum amb les seves zones patològiques segmentades.

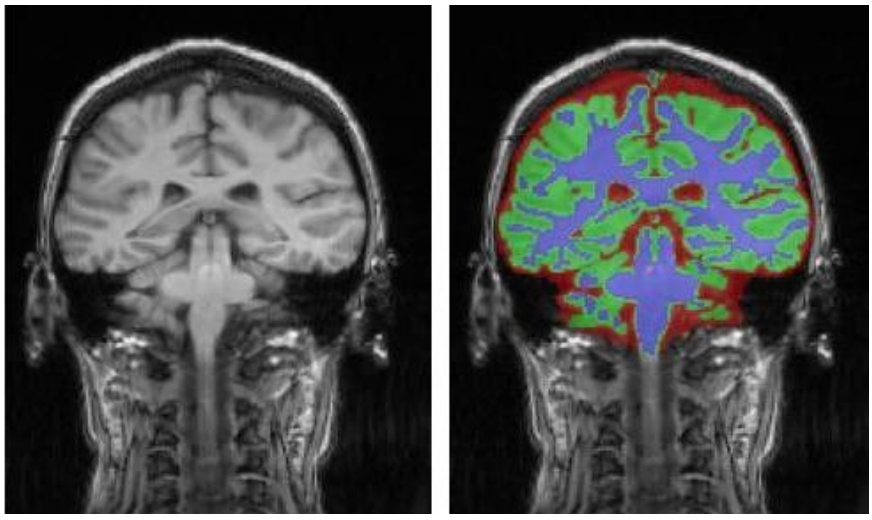


Fig.13 - Imatge RMI del cervell (esquerra), segmentació de la imatge (dreta).

http://cs.adelaide.edu.au/~carneiro/research_MIA.html

2.3 Interacció amb volums

Una altra de les grans necessitats en el processament de volums mèdics RMI-PET per a la medicina nuclear és el fet de poder interactuar a temps real amb la interfície contenidora de la imatge del volum segmentat obtingut.

Com s'ha exposat anteriorment, la renderització de volums plasma en un widget 2D una imatge o volum tridimensional. Sabent això, és evident que és imprescindible el fet de poder interactuar amb aquesta per tal de poder veure el volum mèdic des de totes les posicions i perspectives possibles i no només en la resultant final. D'aquesta manera, podrà detectar-se qualsevol zona patològica que hi hagi en qualsevol part del cos per molt amagada que es trobi en un principi (per exemple, alguna zona que quedi darrere de la càmera i no sigui observable sense interactuar amb la interfície).

El fet de tenir una càmera és essencial per realitzar aquesta interacció ja que permet canviar la vista des d'on s'observa el volum canviant els seus paràmetres (angles de visió, matriu model-view...). Tanmateix, per realitzar aquests canvis a temps real es necessita interactuar amb la interfície.

2.4 Aplicació de partida en MATLAB

La motivació principal d'aquest projecte parteix de les necessitats d'optimització de de la visualització en temps real des d'una aplicació prèvia ja desenvolupada en **MATLAB[11]**.

Aquesta aplicació realitza el renderitzat en la CPU amb un algorisme de visualització de volums anomenat **Shear-Warp[1]** que, tot i donar renderitzats de qualitat, no dóna temps interactius quan tracta màscares de classificació.

L'algorisme Shear-Warp intenta millorar la qualitat visual d'altres algorismes basats només en la profunditat com el Z-Buffer. Els algorismes de Z-Buffer, o de splatting, aproximen la projecció de vòxels amb funcions gaussianes i quan la càmera es situa en angles que no són els eixos coordenats, donen poca qualitat final a les imatges. El Shear-Warp realitza en primer lloc una projecció paral·lela, ordenada segons un dels eixos del volum (**shear**) i després fa una deformació de la imatge obtinguda (**warp**) per obtenir la imatge final ***(veure Fig.14)***. Degut a tot aquest procés, convé realitzar una sèrie de càlculs i processos matricials a MATLAB (en la CPU) on , si a més a més hi sumem la integració de la màscara de segmentació i les limitacions tant de MATLAB com de la CPU, es ralentitza molt el procés de visualització volumètrica donant un alt cost temporal que cal evitar.

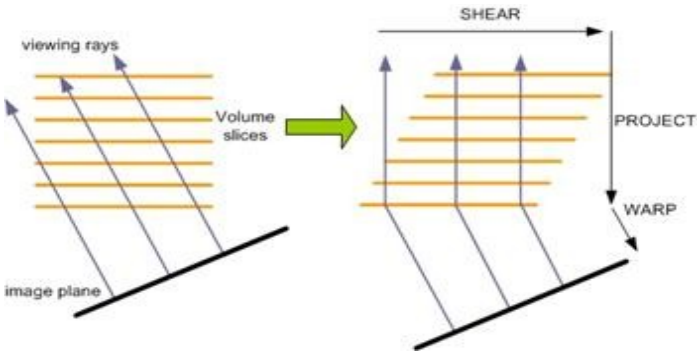


Fig. 14 – Procés Shear-Warp

http://www.byclb.com/TR/Tutorials/volume_rendering/ch1_1.htm

En el present projecte es proposa un algorisme de Ray Casting de volum en mode MIP

implementat a la GPU per a obtenir visualitzacions de qualitat i en temps reals dels volums amb la màscara de segmentació incorporada.

En aquesta aplicació hi ha l'opció de realitzar una segmentació del volum a partir d'una màscara binària per determinar i remarcar les zones possiblement patològiques (amb la limitació referida en *l'apartat 2.2*). Aquestes zones (els uns de la màscara) es pinten de color vermell ***(veure Fig.15)*** un cop es carrega la màscara. La incorporació de la màscara es un procés molt lent que caldrà millorar amb l'algorisme citat.

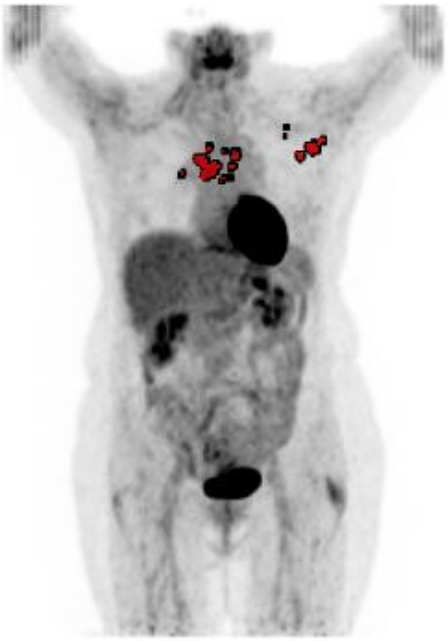


Fig.15 -Mostra volum segmentat amb zones vermelles com a patològiques

L'aplicació conté també una interfície que permet la interacció amb els volums segmentats realitzant un clic de ratolí en una de les zones del widget que el mostra. Un cop clicat, es pot moure el volum rotant-lo amb el ratolí ***(veure Fig.16)***. Degut a aquest tipus d'interacció, es requereix d'una altra màscara de coordenades que emmagatzemi, per a cada píxel, les coordenades del vòxel final pintat (en cas del MIP, el de màxima intensitat). La màscara en qüestió, s'obtindrà com a sortida conjuntament amb la imatge del volum segmentat després d'aplicar la funció del Ray Casting de volum i això permetrà saber les coordenades del vòxel en el píxel clicat.

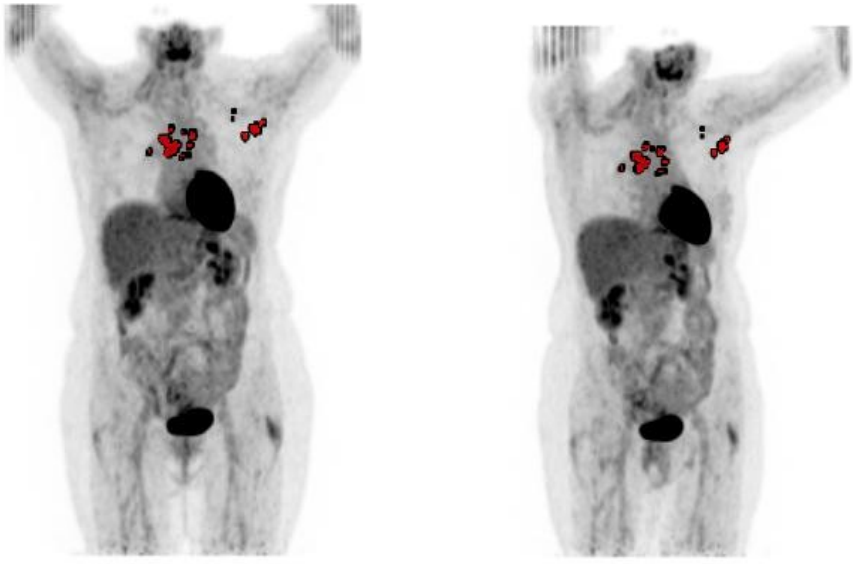


Fig.16 - Volum segmentat, mogut a partir de la interacció

La interacció que permet el moviment del volum és extremadament lenta degut a les propietats de MATLAB. Amb la implementació i integració del Ray Casting de volum en GPU des d'una llibreria de **C++[2]**, s'accelerarà aquest procés fent-lo molt més flexible, àgil de manegar i amb un baix cost temporal.

Addicionalment es permet també a l'usuari poder augmentar o reduir el contrast de la imatge del volum mostrada en la interfície i així ajustar-lo al mode de visió que es desitgi ***(veure Fig.17)***. Per realitzar-ho, tan sols caldrà moure la rodeta o scroll del ratolí. Aquesta utilitat és especialment interessant quan es visualitzen dades procedents de SPECT i PET on els valors contenen molt soroll i és difícil obtenir un llindar per a visualitzar les dades.

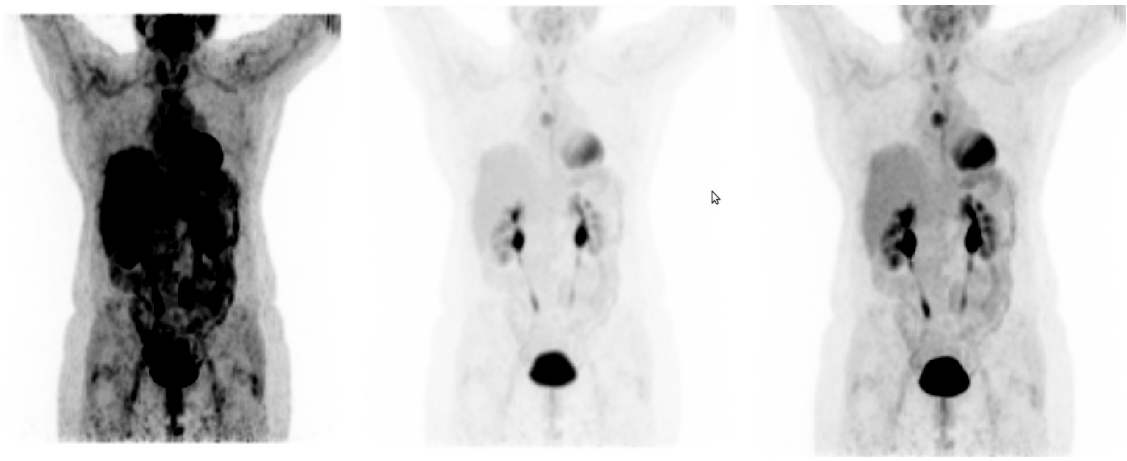


Fig.17 – Imatge del volum amb diferents graus de contrast

2.5 Conclusions

Per a poder processar volums mèdics RMI-PET tridimensionals, és necessari un procés de renderitzat que el transformi de 3D a 2D i el plasmi en un widget.

Un volum és un conjunt organitzat de partícules amb una determinada intensitat anomenades vòxels. El conjunt organitzat d'aquests forma el món de vòxels. Un dels elements imprescindibles per visualitzar volums 3D i poder interactuar amb ells és l'existència d'una càmera que el pugui mostrar des de tots els seus angles de visió possibles.

La tècnica de renderitzat en 3D més útil pels seus bons resultats és la denominada com a Ray Tracing. Aquesta tècnica es basa en l'ús de l'algorisme Ray Casting i per a conjunts de dades tridimensonals la seva adaptació seria l'algorisme Ray Casting de volum. El Ray Casting de volum aplica un traçat de rajos (un per píxel) que llança cap al volum 3D i obté un color RGBA del píxel final amb un determinat càlcul del conjunt de valors que tenen tots els vòxels travessats pel mateix raig. Aquest algorisme obté un rendiment més elevat si es programa sobre la GPU (degut a les propietats d'aquesta) i si s'usa la tècnica *One Step*. Un dels mètodes més emprats per calcular la imatge final a partir de dades mèdiques d'activitat funcional (com són el PET i el SPECT) mitjançant l'algorisme de Ray Casting és el mode MIP, que captura el vòxel de major intensitat de tots els travessats pel mateix raig.

La segmentació de volums és necessària per identificar i separar zones que continguin característiques similars de la resta. Els resultats faciliten la tasca de classificació i és molt útil en àmbits com el la medicina nuclear per establir diagnòstics. Tanmateix, interactuar amb el volum es de gran utilitat per observar totes les seves zones.

En l'aplicació MATLAB de partida, es pot segmentar un volum i diferenciar-ne unes determinades zones patològiques amb l'aplicació d'una màscara binària que pinta els uns d'un color diferent a la resta. Per poder interactuar en temps real amb el volum i observar-lo en la seva totalitat és necessària una màscara de coordenades que conté en cada píxel les coordenades del vòxel capturat en l'aplicació del Ray Casting. El present projecte es centrarà en optimitzar els costos temporals dels processos de càrrega volumètrica amb segmentació incorporada i la interacció d'aquesta aplicació.

3. Anàlisi

En el present capítol s'exposaran els diferents casos d'ús que descriuran els passos a realitzar per usuari extern que vulgui dur a terme un procés o un objectiu determinat en l'interacció amb la interfície de l'aplicació, així com els passos que l'aplicació de MATLAB efectuarà en rebre algun event exterior.

Els casos d'ús possibles, en el nostre context, són tres: inicialitzar el renderitzat del Ray Casting de volum en GPU, actualitzar-lo i modificar tant els paràmetres de la càmera per canviar-lo de punt de vista, com el constrast de les imatges finals.

També es mostrarà el model de domini del projecte que ajudarà a entendre el procés de desenvolupament, l'estructura i la implementació del mateix, que serà explicada amb detall en el *capítol 5*.

El capítol s'estructura doncs en dos subcapítols:

3.1 *Casos d'ús de l'aplicació*
3.2 *Model de domini*

3.1 Casos d'ús de l'aplicació

La interfície d'interacció amb l'usuari està realitzada mitjançant MATLAB. Aquesta, conté una pluralitat d'opcions i funcionalitats diverses dins les diferents barres desplegables dels seus menús. Aquest projecte només es centra en la càrrega i renderitzat de volums mèdics segmentats utilitzant de forma externa l'algorisme Ray Casting en volum i la interacció amb el resultat d'aquesta visualització des de l'aplicació de MATLAB. Per tant, només ens interessarà una petita part de la funcionalitat d'aquesta interfície, i el casos d'ús mostrats seran els referents a la temàtica en què es centra aquest projecte.

En conseqüència, els tres casos d'ús a analitzar són els següents:

1. Inicialitzar el renderitzat del volum segmentat a partir de l'algorisme Ray Casting en volum.
2. Actualitzar la visualització del volum a partir de la interacció per modificar els paràmetres de la càmera i canviar el punt de vista del volum.
3. Actualitzar la visualització de volum modificant el contrast de la imatge final.

En els tres casos d'ús, s'han detallat les accions distingint-les si són controlades pel sistema o si s'han controlat des de l'aplicació de MATLAB.

3.1.1 Cas d'ús 1 (UC1)

Inicialitzar el renderitzat del volum segmentat a partir de l'algorisme Ray Casting en volum.

Actor principal: Usuari de l'aplicació de MATLAB

Personal involucrat i interessos: L'usuari de l'aplicació voldrà visualitzar un determinat volum RMI-PET segmentat, utilitzant l'algorisme del Ray Casting de volum. L'aplicació MATLAB voldrà cridar la funció que realitza aquest renderitzat.

Precondicions: L'usuari haurà obert prèviament l'aplicació. Posteriorment haurà carregat

un dels volums RMI-PET continguts en l'aplicació a partir de l'opció del menú pertinent. Seguidament, haurà hagut d'activar la màscara de segmentació amb l'opció del menú de “Show mask” (mostrar la màscara de segmentació) i haver triat l'opció de render “MIP GPU” .

Escenari principal:

1. L'usuari activa el menú de l'aplicació MATLAB que cridarà la funció del renderitzat en Ray Casting de volum a la GPU en mode MIP, on se li passaran els paràmetres necessaris (volum carregat, mida volum, màscara carregada, mida imatge...).
2. El sistema utilitza la funció anterior i crida mètodes d'una llibreria externa que obrirà un widget iconificat, inicialitzarà el context de GL, compilarà i linkarà els shaders i realitzarà l'algorisme de Ray Casting de volum en la GPU, al fragment shader.
3. L'aplicació MATLAB recull la lectura dels framebuffers obtinguts en llegir els píxels del fragment shader.
4. L'aplicació MATLAB omple les matrius de sortida de la imatge del volum segmentat a mostrar i de la màscara de coordenades, amb el contingut dels buffers anteriors.
5. L'aplicació MATLAB mostra la imatge resultant en la seva interfície d'interacció.

Postcondicions: El resultat final dependrà dels valors de la càmera, la mida del món de vòxels del volum triat i les zones patològiques de la màscara de segmentació associada.

3.1.2 Cas d'ús 2 (UC2)

Actualitzar la visualització del volum a partir de la interacció per modificar els paràmetres de la càmera i canviar el punt de vista del volum.

Actor principal: L'usuari de l'aplicació de MATLAB

Personal involucrat i interessos: L'usuari de l'aplicació voldrà visualitzar un volum RMI-

PET segmentat concret després d'haver interactuat amb ell per moure'l i veure'l des d'un altre punt de vista. L'aplicació voldrà actualitzar el renderitzat en Ray Casting del volum segons la interacció rebuda.

Precondicions: L'aplicació MATLAB ja haurà mostrat almenys una vegada el volum RMI-PET segmentat amb l'algorisme de Ray Casting de volum. L'usuari haurà hagut d'interactuar amb la interfície fent un clic de ratolí i un petit moviment a la zona on es mostra la imatge del volum.

Escenari principal:

1. L'usuari refresca la càmera o selecciona el mode de rotació contínua
2. L'aplicació MATLAB crida la funció del renderitzat en Ray Casting de Volum i li passa els paràmetres necessaris (volum carregat, mida volum, màscara carregada, mida imatge...).
3. El sistema penetra en la funció tot anant cap el mètode d'una llibreria externa que actualitza la visió del volum segmentat canviant només els paràmatres de la càmera. Després, toranarà a activar els shaders per recaulcuar el Ray Casting de volum.
4. L'aplicació MATLAB recull la nova lectura dels framebuffers obtinguts en llegir els píxels del fragment shader.
5. L'aplicació MATLAB omple les matrius de sortida de la imatge del volum segmentat a mostrar i de la màscara de coordenades amb el contingut dels buffers anteriors.
6. L'aplicació MATLAB mostra la nova imatge resultant en la seva interfície d'interacció.

Postcondicions: El resultat final dependrà dels valors de la càmera, la mida del món de vòxels del volum triat i les zones patològiques de la màscara de segmentació associada.

3.1.3 Cas d'ús 3 (UC3)

Actualitzar la visualització de volum modificant el contrast de la imatge final.

Actor principal: L'usuari de l'aplicació de MATLAB

Personal involucrat i interessos: L'usuari de l'aplicació voldrà canviar (augmentar o reduir) el contrast del volum RMI-PET segmentat que es veu en la interfície. L'aplicació MATLAB voldrà actualitzar el contrast de la imatge del volum segons la interacció de l'usuari.

Precondicions: L'usuari ja haurà carregat un volum dels existents. L'aplicació MATLAB tindrà la imatge del volum (independentment de si està segmentat o no i de si es visualitza per Ray Casting o no) en la seva interfície.

Escenari principal:

1. L'usuari mou la rodeta (scroll) del ratolí en una direcció (amunt o avall) per modificar el contrast de la imatge del volum projectada.
2. L'aplicació MATLAB rep l'event, va a la funció d'actualització del contrast i segons la direcció del moviment, disminuirà (moviment amunt) o augmentarà (moviment avall) un factor de contrast en la imatge.
3. L'aplicació MATLAB actualitza la imatge projectada del volum amb el canvi de contrast pertinent en la seva interfície.

Postcondicions: El resultat final dependrà de les vegades que l'usuari hagi volgut canviar el contrast, de si l'ha volgut augmentar o disminuir, i del factor d'increment/decrement de contrast per moviment de rodeta del ratolí.

3.2 Model de domini

El model de domini següent mostra les entitats (amb els atributs del seu constructor i les relacions) utilitzades en la llibreria externa que implementa l'algorisme de Ray Casting de volum, juntament amb l'entitat genèrica representant de la interfície gràfica de l'aplicació de MATLAB, i l'entitat dels shaders (representant del procés realitzat en GPU) ***(veure Fig.18)***.

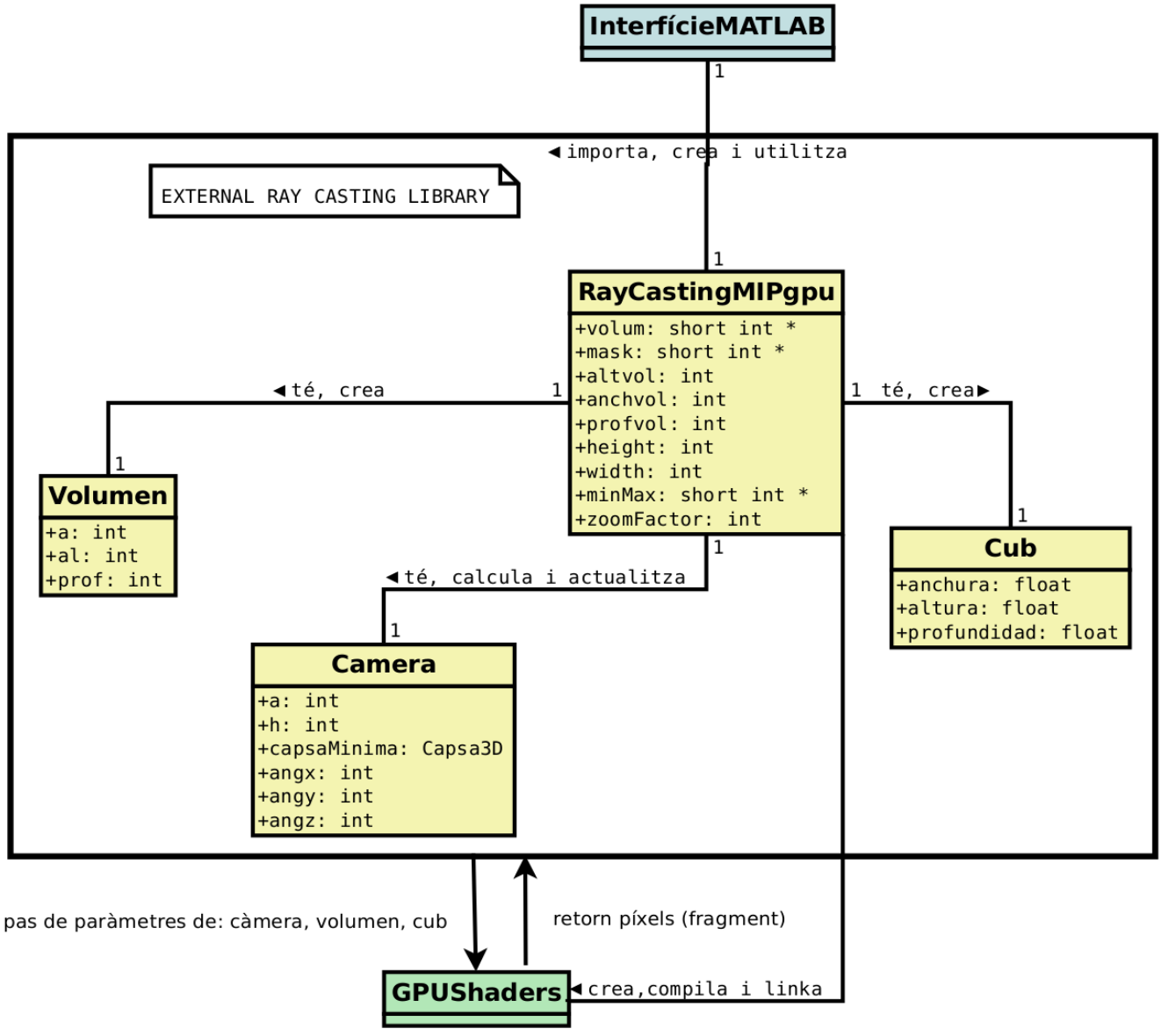


Fig-18 – Model de domini de l'aplicació

Descripció de les entitats:

- ***InterfícieMATLAB:*** És l'entitat que representa la interfície gràfica generada en MATLAB que mostra la imatge del volum carregat mitjançant alguna tècnica de renderitzat i amb la qual l'usuari pot interactuar. És l'encarregada de cridar les funcions de renderitzat, entre elles, l'anomenada i implementada Ray Casting de volum. Entre altres coses, i a partir del procés que es descriurà en *l'apartat 5.1* de la implementació, és la responsable de connectar amb la llibreria externa que implementa l'algorisme Ray Casting sobre la GPU i rebre i mostrar-ne els resultats.

- Les classes: *RayCastingMIPgpu*, *Volumen*, *Camera* i *Cub* són les utilitzades en la llibreria externa que realitza el renderitzat de Ray Casting de volum en mode MIP sobre la GPU i que connecta amb la interfície de MATLAB.

  - ***RayCastingMIPgpu:*** És la classe principal de la llibreria. Podria assumir-se com l'entitat respresentativa de l'escena de la nostra implementació. Crea i conté una càmera, a la que inicialitza i actualitza els seus paràmetres, un volum *Volumen* que li serveix per carregar i processar els vòxels d'aquest i la seva màscara de segmentació, i un *Cub*, útil per realitzar la tècnica ja descrita del *One Step*, passar les dades a la GPU, i dibuixar i capturar els buffers de píxels resultants de la GPU. També s'encarrega d'inicialitzar els paràmetres de GL i de crear, compilar i linkar els shaders de la GPU i passar-los-hi les dades necessàries (dades volum, saturació...), per tal d'interactuar amb la GPU i obtenir el resultat d'aplicar el Ray Casting de volum en mode MIP.

  - ***Volumen:*** Classe utilitzada per carregar i processar els vòxels del volum i la màscara de segmentació associada. S'encarrega d'omplir el buffers corresponents amb les dades de la màscara i del volum per poder passar-los a la GPU com a textures i treballar amb ells aplicant-hi l'algorisme Ray Casting. És crea un únic volum en *RayCastingMIPgpu*.

- ***Camera:*** La classe *Camera* és la que representa l'entitat de la càmera. Conté una sèrie de paràmetres (angles de visió, matriu model-view, projecció...), que permetran que es projecti el nostre volum en l'escena del widget final amb unes característiques de visió determinades . Una única càmera es crea en *RayCastinMIPgpu* els atributs de la qual es calculen, inicialitzen, actualitzen i passen a la GPU cada vegada que l'usuari interactua amb la interfície.

- ***Cub:*** Classe necessària per poder realitzar el Ray Casting de volum sobre la GPU en un sol pas "*One Step*" (explicat en *l'apartat 2.1.6.1* dels antecedents). S'encarrega de fer el pas a la GPU (shaders) dels punts, els colors i les textures del cub i de dibuixar i capturar els buffers dels píxels de la imatge final i de les coordenades rebuts del fragment shader després d'aplicar el Ray Casting utilitzant aquesta informació. El cub es crea en la classe *RayCastingMIPgpu* que rebrà els seus buffers de retorn .

- ***GPUShaders:*** És l'entitat que representa el procés que es duu a terme en la GPU. Per interactuar amb aquesta, es tenen els "shaders" programats en GLSL (fragment shader i vertex shader, descrits en *l'apartat 2.1.6* dels antecedents). La *RayCastingMIPgpu* crea el programa per utilitzar els shaders, compilar-los i linkar-los. Posteriorment es fa un pas de dades cap a la GPU (paràmetres de la càmera, saturació, textura del volum, textura de la màscara ...etc), que són necessaris per l'algorisme del Ray Casting de volum en mode MIP realitzat en el fragment shader. Aquest, retorna dos buffers de píxels resultants de l'algorisme (els de la imatge i els de les coordenades) que són capturats i retornats cap a MATLAB on es farà el procés necessari per mostrar la imatge en la interfície.

Tot el funcionament intern de les entitats serà explicitat i detallat en la seva totalitat tant en *l'apartat 4* referent al disseny com en *l'apartat 5* referent a la implementació

4. Disseny

Al llarg d'aquest capítol es mostrarà i explicarà el disseny de l'aplicació a partir del diagrama de classes, que s'estendrà del model de domini mostrant també els mètodes i atributs de cada classe. També s'efectuaran els diagrames de seqüència derivats dels tres casos d'ús detallats en l'apartat anterior.

El captítol s'estructura amb els següents subapartats:

4.1 *Diagrama de classes de l'aplicació*

4.2 *Diagrames de seqüencia*

4.1 Diagrama de classes

El diagrama de classes que s'il·lustra a continuació ***(veure Fig.19),*** conté les mateixes entitats que el model de domini mostrat en el capítol anterior, però ara, no només es detallaran els atributs del constructor de cada classe o entitat, sinó que també es mostren les seves operacions o mètodes interns amb els paràmetres i el retorn de cada un, i els atributs genèrics de cada classe.

Per fer-lo un pèl més explícit i aclaridor que el model de domini, l'entitat que representa el procés sobre la GPU (GPUShaders), es dividirà en dues entitats: el fragment shader i el vertex shader. D'aquesta manera es veurà més clarament com es passen i obtenen dades dels shaders de la GPU, i tindrem una visió molt més concreta i realista sobre la implementació.

S'han afegit també com a entitats tant la MEX-function de C++ que serveix d'interfície entre MATLAB i el renderitzat en Ray Casting de Volum mode MIP sobre la GPU i utilitza la llibreria externa, com el *header (.h)* que connecta amb OpenGL. Per crear un context de GL és necessari obrir un widget addicional. Aquest widget addicional és redundant amb l'aplicació MATLAB però és necessari per poder activar la GPU. Així que s'ha optat per obrir-lo en forma d'icona, transparent a l'usuari final de l'aplicació.

Les MEX-function[22] no són realment classes, ja que MATLAB no és un llenguatge orientat a objectes. No obstant, el fet de posseir d'una sèrie de mètodes interns i uns atributs, doten les MEX-functions de les mateixes característiques d'una classe de C++. En conseqüència, poden ser considerades com a tal dins el context de MATLAB .

Finalment, s'explicarà de forma abreujada cada una de les entitats aquí mostrades, els seus atributs i els seus mètodes, per tal de tenir una idea prèvia al que s'exposarà en el següent capítol de la implementació i el desenvolupament intern, on s'entrarà més en detall sobre el funcionament *(capítol 5)*.

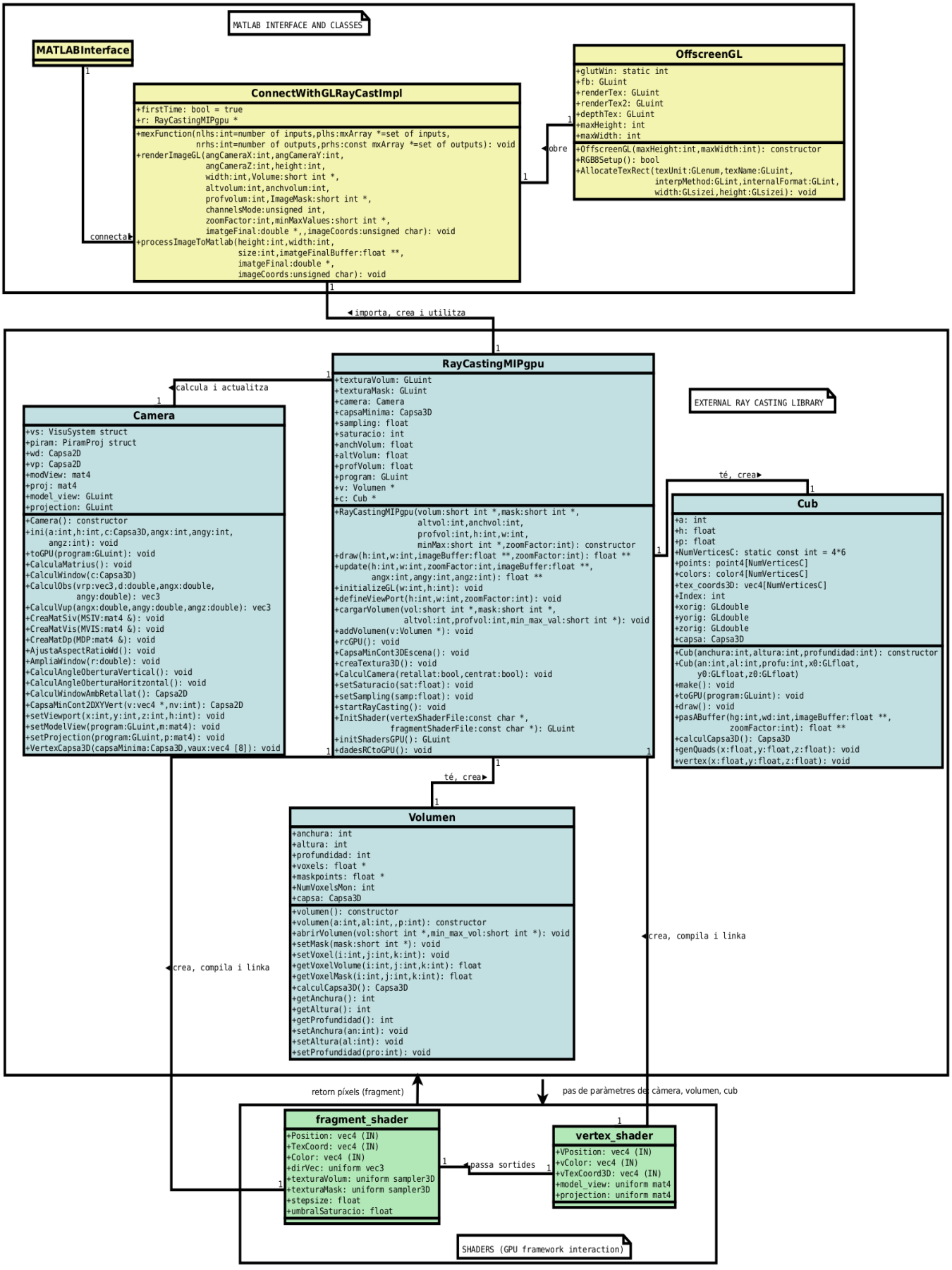


Fig.19 – Diagrama de classes de l'aplicació

- *Les classes de color blau són les que pertanyen a la llibreria externa que implementa l'algorisme de Ray Casting de volum.*
- *Las classes de color groc formen part de MATLAB.*
- *Les entitats verdes representen els shaders de la GPU, que es comuniquen amb la llibreria, però estan inclosos en la carpeta dels arxius de MATLAB.*

Seguidament es farà una descripció individual de cada una de les classes o entitats mostrades en el diagrama ***(veure Fig. 20, 21, 22, 23, 24, 25, 26 i 27).***

Fig. 20 – Detall Interfície de MATLAB

Aquesta entitat representa la interfície gràfica de l'aplicació de MATLAB. És on es mostrarà el resultat final del volum després d'haver efectuat el Ray Casting de volum sobre la GPU i en mode MIP. També servirà per interactuar amb l'usuari i poder actualitzar la visualització des de diferents punts de vista.

No existeix dins del codi, ja que és una abstracció per a representar la interfície gràfica, i com a tal, no se li adjudiquen ni atributs ni mètodes. En realitat, s'executaria a partir d'un script ".m" de MATLAB. La interfície real conté un ventall d'opcions i funcionalitats molt ampli. En el present projecte ens centrem només en la crida a la funció que realitza el render en Ray Casting sobre la GPU, que és el que ens interessa. Per tant, com es pot comprovar en el diagrama de classes, la interfície de MATLAB crida la funció MATLAB del renderitzat en Ray Casting MIP GPU, i connecta amb *ConnectWithGLRayCastImpl* que és la MEX-function que la implementa i li donarà els resultats a mostrar utilitzant la llibreria externa de l'algorisme del renderitzat.

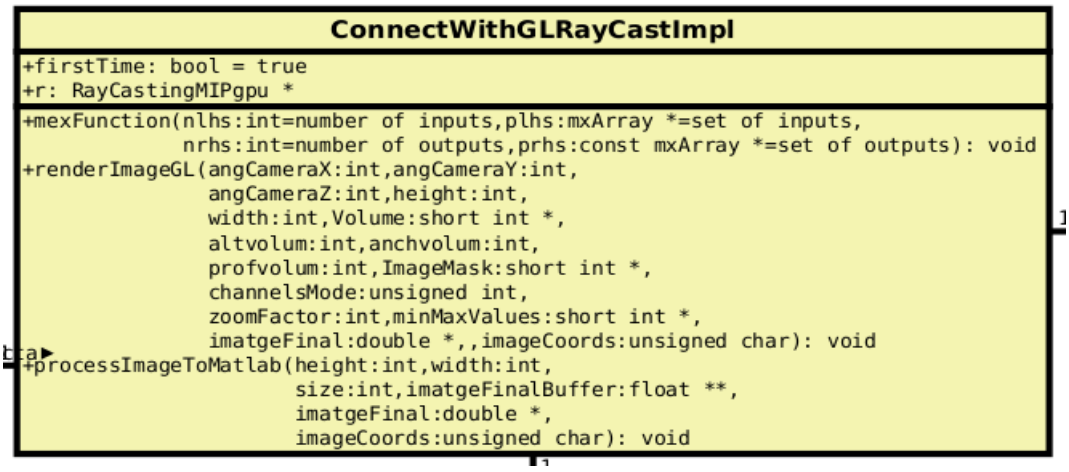


Fig. 21 – Detall MEX-Function *ConnectWithGLRayCastImpl*

És la MEX-function citada anteriorment. Es troba dins de MATLAB i permet implementar codi C++ des d'aquest. Dins el context de MATLAB actua com una classe, ja que posseeix una sèrie de mètodes i atributs.

La MEX-function és la peça clau que estableix totes les connexions i inicialitzacions necessàries amb OpenGL i la llibreria del renderitzat en Ray Casting de Volum. Importa i utilitza la llibreria i n'instancia un objecte per tal de cridar els seus mètodes. Posseeix els mètodes adients *"renderImageGL()"* per aplicar el renderitzat a partir de la crida de la llibreria i per processar el resultat d'aquesta i tornar-lo en el format adequat cap a la interfície de MATLAB *"processImageToMatlab()"*. És la que distingirà, a partir d'un booleà, entre la primera vegada que es crida la funció del render i la resta de vegades, on ja només caldrà dur a terme una actualització.

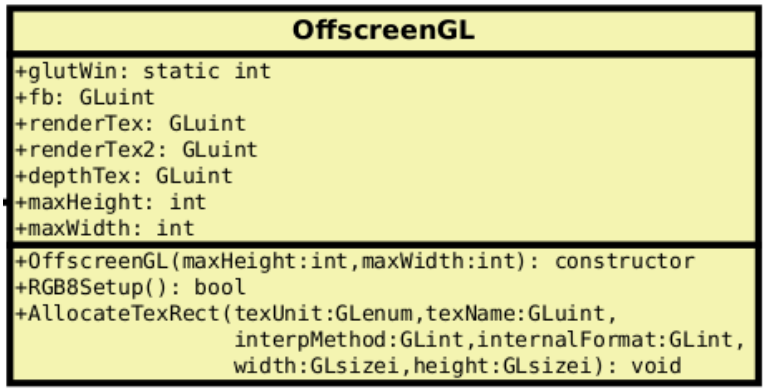


Fig. 22 – Detall classe *OffScreenGL*

La classe *OffScreenGL* és una classe de C++ continguda i implementada en un arxiu ".h" *(header)* perquè pugui ser cridada i importada amb facilitat en MATLAB.

És utilitzada i instanciada per la MEX-function mostrada prèviament. Permet la connexió amb OpenGL, la inicialització dels paràmetres d'aquesta i la creació i obertura d'un widget GL iconificat per poder efectuar la captura dels píxels de sortida de la GPU sense problemes, tal com s'ha explicat anteriorment (*veure apartat 4.1*) . Ha de ser utilitzada abans de la crida a la llibrera, ja que no es pot aplicar l'algorisme de Ray Casting sobre la GPU si abans no es connecta amb OpenGL que proveeix de les eines adients per a poder treballar sobre aquesta.

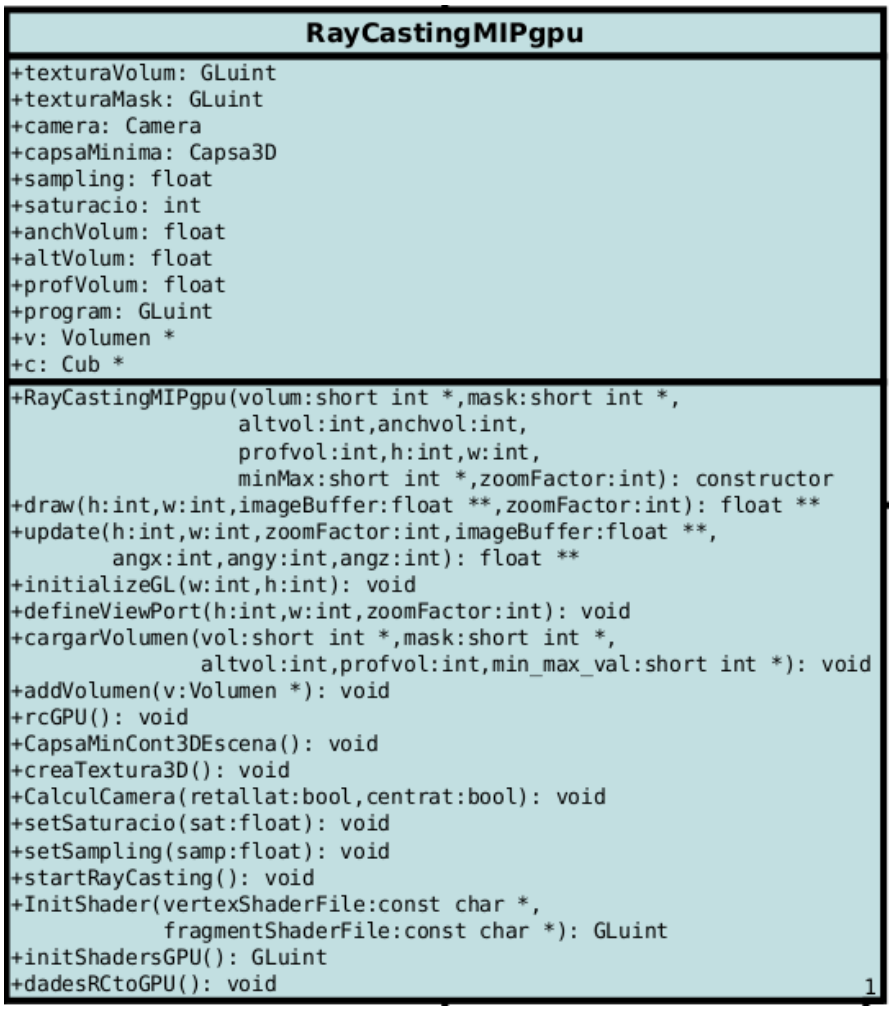


Fig. 23 – Detall classe *RayCastingMIPgpu*

RayCastingMIPgpu és la classe principal de la llibreria externa que durà a terme el Ray Casting de volum sobre la GPU en mode MIP i oferirà els resultats obtinguts a la interfície de MATLAB. S'importa i es crida des de la MEX-Function de la ***Fig.20***.

Realitza la major part del treball ja que és la contenidora i creadora de tots els objectes necessaris per dur a terme el render sobre la GPU. Reuneix un conjunt d'atributs i mètodes que li confereixen poder realitzar les seves funcions.

Les seves funcions són tantes i tan diverses com (*procés detallat en el capítol 5.3*):

- Inicialitzar paràmetres de GL i definir el viewport (que ho realitza en el seu constructor) i calcular la capsa mínima contenidora de l'escena.
- Carregar el volum provinent de MATLAB i la seva màscara de segmentació, crear una classe *Volumen* i cridar els mètodes necessaris per processar-ne els seus vòxels i passar-los cap a la GPU per ser utilitzats.
- Crear una càmera i a partir dels mètodes compresos en ella, definir els seus paràmetres, calcular-los, actualitzar-los i passar-los cap a la GPU.
- Crear el programa que contindrà els shaders de la GPU. Compilar-los, linkar-los i accedir a ells perquè la GPU realitzi el Ray Casting de volum en mode MIP a partir de les dades passades.
- Crear una classe *Cub* per dur a terme la tècnica ja explicada *One Step* i fer la creació i pas de paràmetres necessaris d'aquest cap a la GPU per poder-ho realitzar.
- Fer el dibuixat i actualització *draw()* i *update()* de la imatge del volum segmentat a partir de les dades del cub, la càmera, el volum, la màscara i altres paràmetres (saturació, sampling...) mitjançant els shaders de la GPU i l'algorisme Ray Casting de volum. Agafar els buffers resultants i retornar-los al punt de crida de MATLAB.

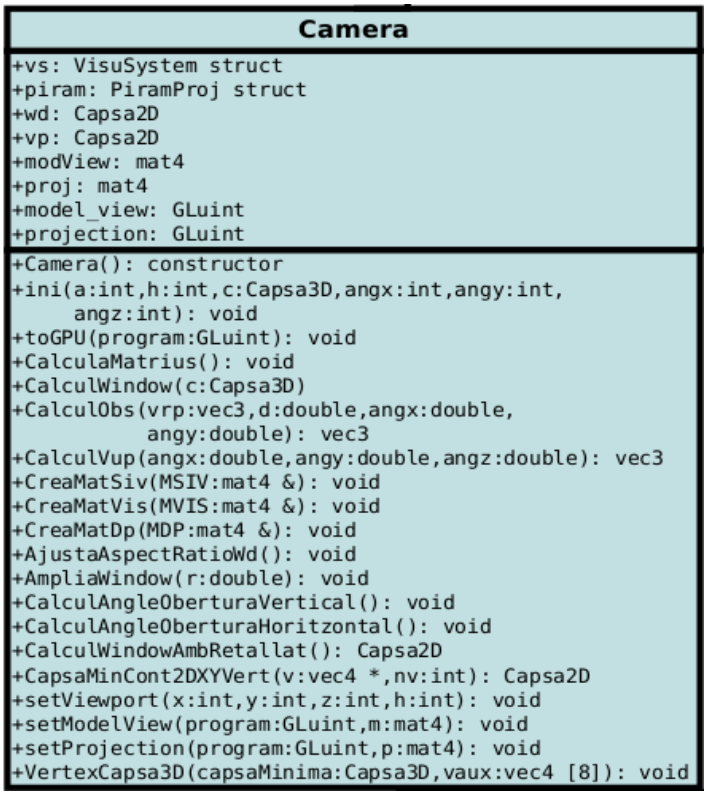


Fig. 24 – Detall classe *Camera*

És la representació de la càmera que farà possible la visualització de qualsevol objecte en l'escena. Es creada per la classe *RayCastingMIPgpu* que la utilitza i crida els seus mètodes.

Conté tots els atributs i mètodes necessaris per a poder calcular els seus paràmetres, canviar-los, obtenir-los i passar-los a GPU (angles de visió, capsa contenidora, observador, VRP, aspect ratio, finestra, viewport, matriu model-view, matriu de projecció...). També permet establir el tipus de projecció (paral·lela o perspectiva) i realitzar (si es vol) un clipping (retallat) a l'escena.

La utilitat dels paràmetres i termes citats en el paràgraf anterior ve detallada i definida en el *capítol 2.1.3* dels antecedents, que defineix i explica el model càmera.

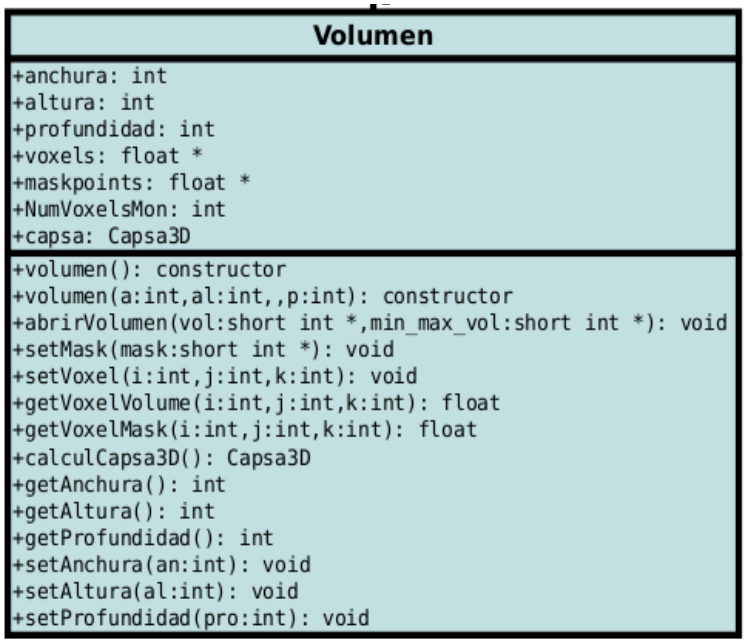


Fig. 25 – Detall classe *Volumen*

La classe *Volumen* és la classe que conté els mètodes i atributs necessaris per dur a terme la càrrega tant del volum com de la màscara de segmentació d'aquest, processar-ne els vòxels i emmagatzemar-los. Entre altres paràmetres, rep el contingut del món de vòxels del volum , la màscara, i la seva mida (ample,alt i profunditat) proporcionats des de MATLAB. S'encarrega també de realitzar un escalat en el processament dels seus vòxels per tal de no perdre qualitat en la imatge final (*detalls en els apartats 5.2 i 5.3 del capítol 5*).

La classe *RayCastingMIPgpu* s'encarrega d'instanciar-ne un objecte i fer la crida als mètodes pertinents d'aquesta per gestionar adequadament tot el procés del pas de les dades del volum i la màscara cap als shaders de la GPU i aplicar-hi l'algorisme del Ray Casting per capturar-ne el resultat final.

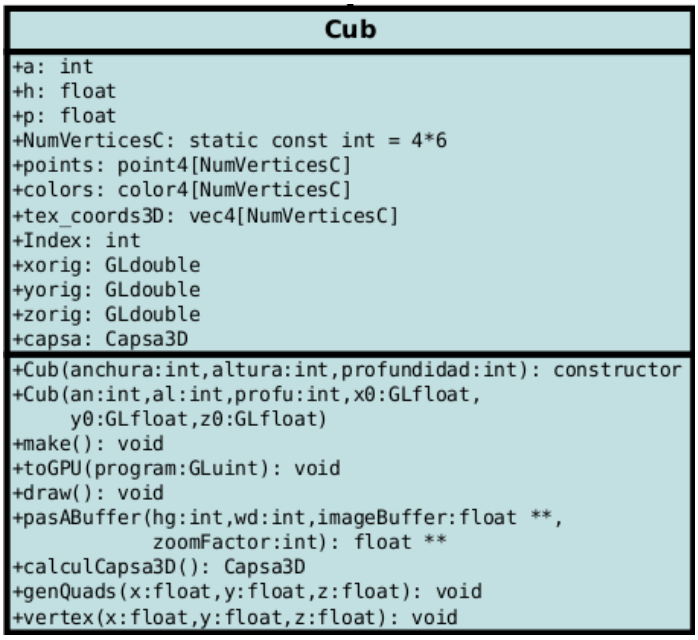


Fig. 26 – Detall classe *Cub*

La classe *Cub* és la classe utilitzada per a poder implementar en un sol pas *One Step* l'algorisme del Ray Casting de volum sobre la GPU i així optimitzar molt més el cost temporal (*procés descrit en l'apartat 2.1.6.1 i en el 5.3*). El *Cub* conté els atributs i mètodes essencials per obtenir el retorn dels buffers de píxels dels shaders de la GPU un cop aplicat el Ray Casting mode MIP amb les dades passades. És el mètode *pasABuffer()* el que fa aquest pas de captura de les sortides del fragment shader de la GPU, un cop passat tot el flux intern d'execució de la llibreria. Serà aquesta sortida la que finalment es rebrà al punt de crida de MATLAB i es mostrarà en la seva interfície gràfica.

Es crea un *Cub* des de la classe *RayCastingMIPgpu* que realitzarà la crida als mètodes pertinents per obtenir els resultats descrits anteriorment i passar les dades del cub requerides cap als shaders de la GPU.

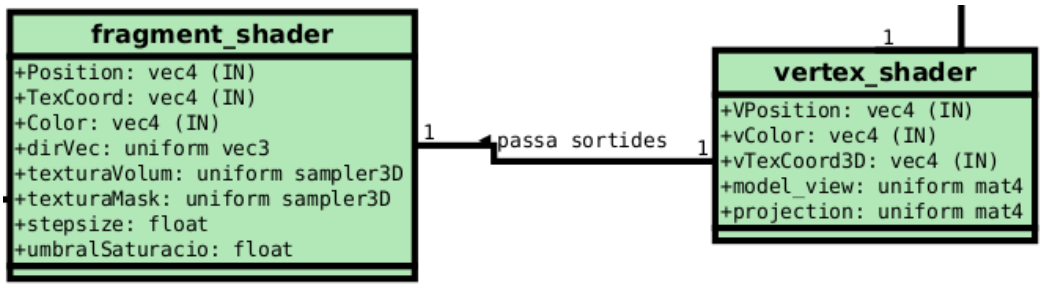


Fig. 27 – Detall Vertex Shader i Fragment Shader

El fragment shader i el vertex shader són els elements que ofereix el framework d'OpenGL que permeten treballar a nivell de GPU. Es creen, compilen i linken independentment i estan implementats en llenguatge d'ombrejat GLSL (més detalls *en l'apartat 2.1.6 dels antecedents*).

És la classe *RayCastingMIPgpu* la que realitza la seva creació a partir d'un "program", i posteriorment el seu compilat, linkat i pas de paràmetres. Per poder accedir a ells necessita determinar la seva ubicació especificant per paràmetre la carpeta on es troba cada un d'ells. Ambdós shaders es troben dins la carpeta que conté els arxius i classes de MATLAB per facilitar-ne la seva ubicació. Tant el vertex shader com el fragment shader treballen usant el paral·lelisme propi de la GPU i reben paràmetres i dades des de diferents mètodes de les classes de la llibreria esmentada.

El vertex shader rep com a entrades (IN) els paràmetres del cub (posicions, colors i coordenades de les textures) i les matrius de visió i projecció de la càmera. És l'encarregat de calcular les transformacions afins necessàries a partir de les matrius de la càmera per establir la projecció adequada del volum cap a la imatge 2D final i fer els canvis de sistemes de coordenades pertinents. Les seves sortides (OUT), passen a ser les entrades interpolades a nivell de píxel en el fragment shader.

El fragment shader és el que duu a terme l'algorisme del Ray Casting de volum en mode MIP. Treballa per píxels i utilitzant les dades del volum '*texturaVolum*', la màscara de segmentació '*texturaMask*', un llindar de saturació i les dades del *Cub* (pel *One Step*). Implementa l'algorisme de renderitzat citat i pinta en cada píxel el valor RGBA obtingut.

Les seves sortides són els píxels amb el color RGBA establert després del Ray Casting en mode MIP. Per garantir una possible selecció 3D posterior, s'obtindran també les coordenades del vòxel capturat. Aquestes dues sortides es recolliran en dos buffers diferents que rebrà el mètode *pasABuffer()* de la classe *Cub* de la llibreria, i aquesta, els enviarà cap a MATLAB per projectar-los en la seva interfície (tot aquest procés es descriurà amb més detall *en el capítol 5*).

4.2 Diagrames de seqüència

En aquest apartat es mostraran els tres diagrames de seqüència derivats dels casos d'ús formulats en el *capítol 3* referent a l'anàlisi. Els diagrames donaran una visió clara i entenedora sobre la interacció i comunicació interna que segueixen el conjunt de funcions i objectes per posar en marxa el procés que es descriu en el cas d'ús a realitzar.

Reprenent els casos d'ús anteriors, tindrem els següents diagrames de seqüència:

1. DS1: Inicialitzar el renderitzat del volum segmentat a partir de l'algorisme Ray Casting en volum.
2. DS2: Actualitzar la visualització del volum a partir de la interacció per modificar els paràmetres de la càmera i canviar el punt de vista del volum.
3. DS3:Actualitzar la visualització de volum modificant el contrast de la imatge final.

** Alguns dels diagrames de seqüència estan modularitzats per fer més fàcil la llegibilitat ja que són massa extensos per poder-los encabir en l'ample de pàgina actual. Per a més detalls, caldrà mirar el capítol següent.*

** No es mostra la totalitat de la implementació i interacció d'algunes parts (càlculs de paràmetres de la càmera, creació del cub...) degut a que són molt extenses i poc rellevants en el present projecte . Tampoc es mostra els tipus de cada paràmetre d'un mètode ja que això ja ho explicita el diagrama de classes. D'aquesta manera es simplifica el diagrama fent-lo més entenedor.*

4.2.1 Diagrama de seqüència 1 (DS1)

Inicialitzar el renderitzat del volum segmentat a partir de l'algorisme Ray Casting en volum.

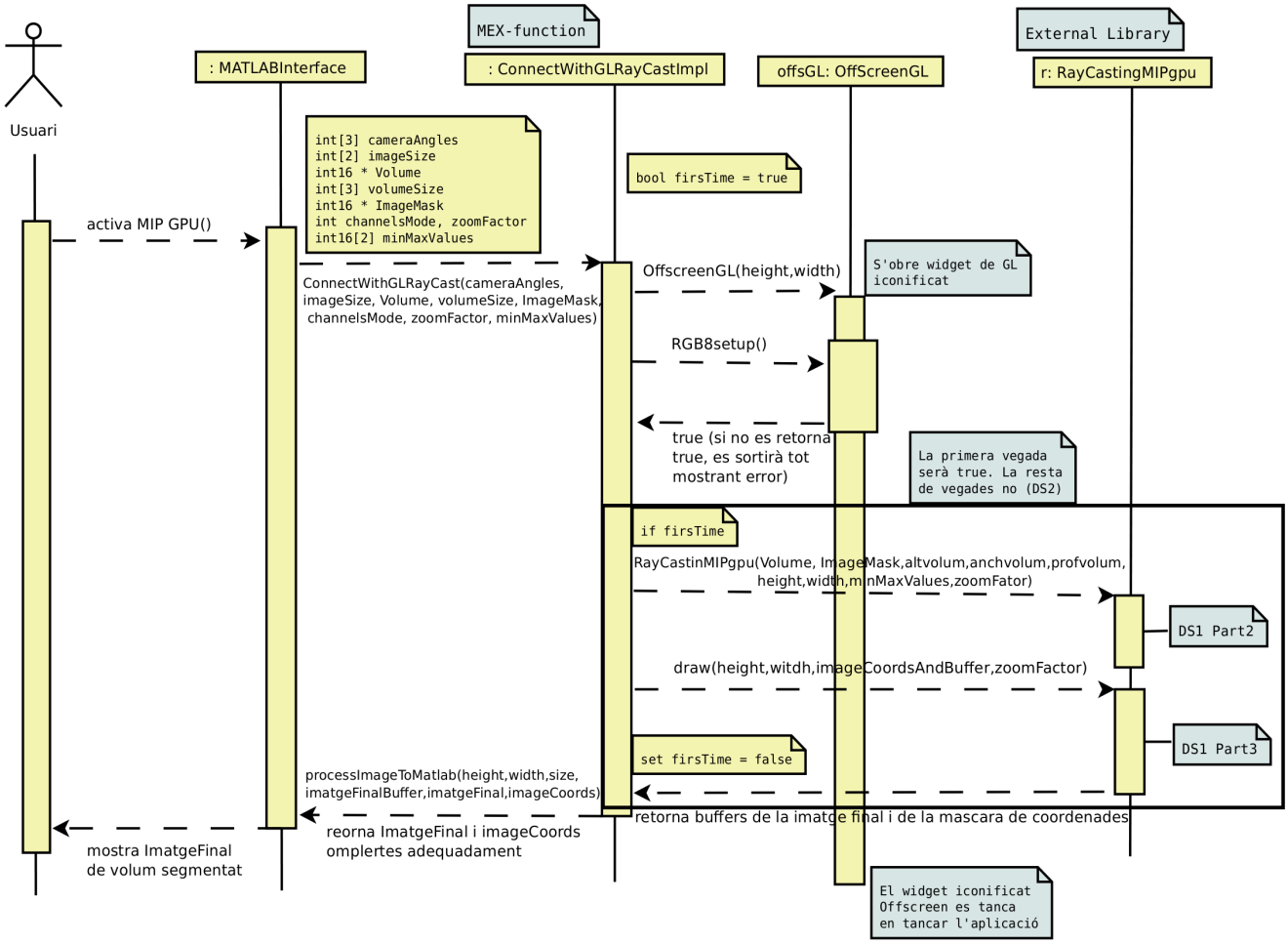


Fig. 28 – DS1 Part1

En aquesta primera part de diagrama es veu com es duu a terme la connexió MATLAB amb OpenGL i la posterior crida a mètodes de la llibreria externa (sent la primera vegada que s'executa) per tal que realitzi l'algorisme de Ray Casting de volum i en retorni la imatge a ser mostrada i la màscara de coordenades.

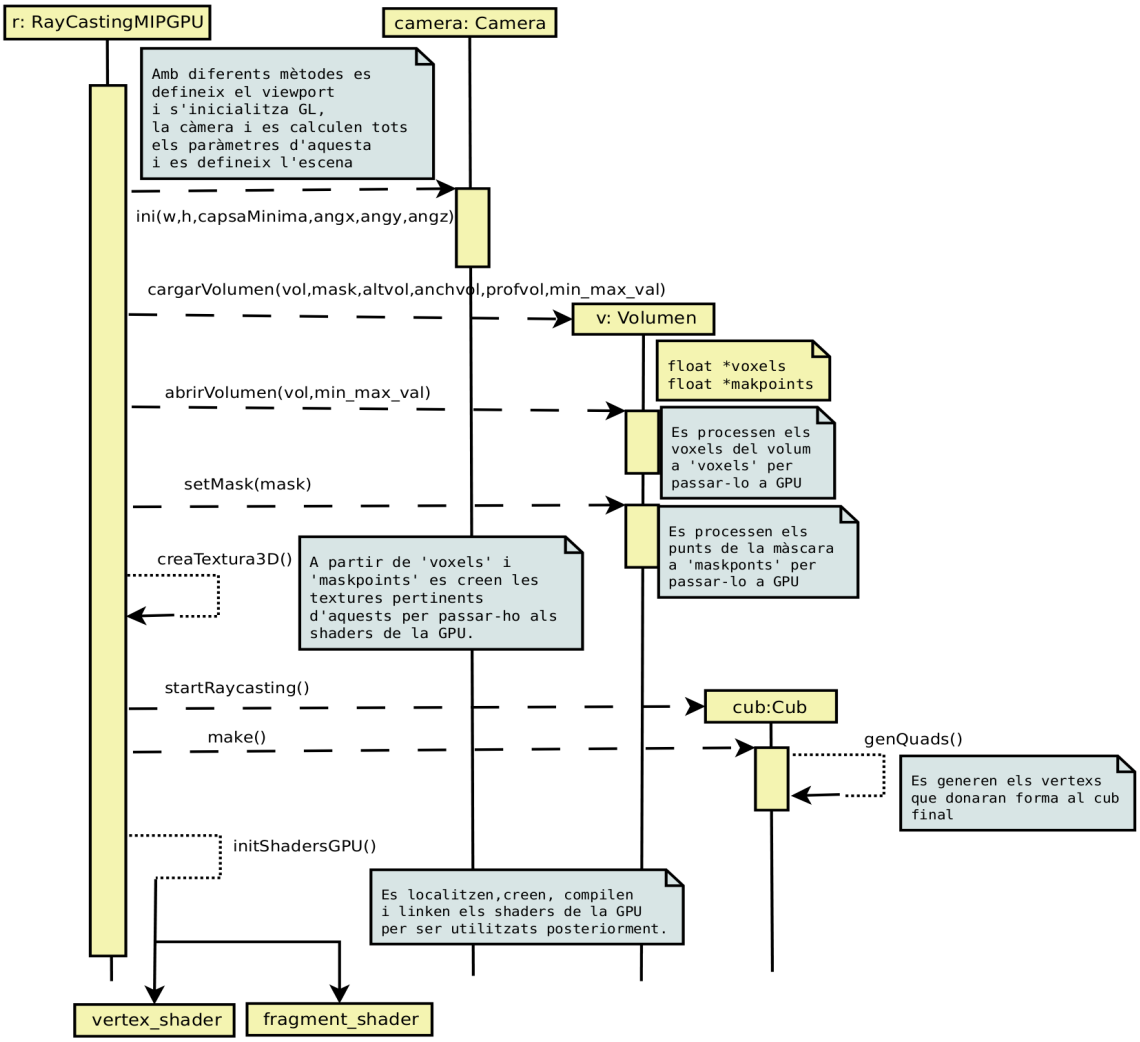


Fig. 29 – DS1 Part 2

En aquesta part del diagrama de seqüència 1, hi ha tot el procés que es realitza en cridar al constructor de la classe principal de la llibreria externa, que només s'executarà la primera vegada que es cridi la funció. Es duen a terme les inicialitzacions de GL, la càmera i els càlculs dels paràmetres d'aquesta i l'escena. Es fa la càrrega del volum i la màscara de segmentació, el processat dels seus vòxels i la creació de les textures a partir d'aquests, per després poder passar a GPU. També es crea el *Cub* i es generen els seus vèrtexs i punts que després es passaran als shaders per aplicar el *One Step*. Finalment s'inicialitzen, compilen i linken els shaders de la GPU per poder ser utilitzats.

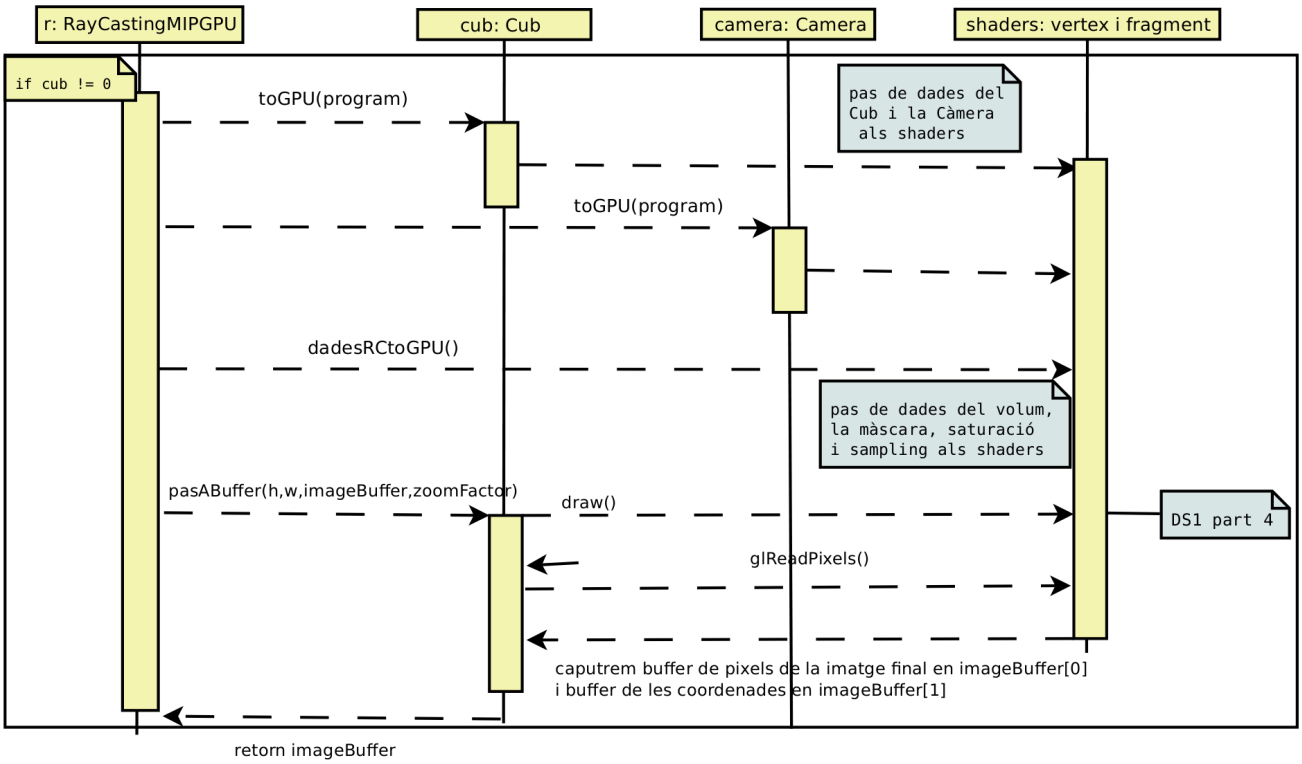


Fig. 30 – DS1 Part 3

En aquesta segona part del primer diagrama de seqüència 1, es mostra tot el procés de dibuixat que es duu a terme després de la crida al constructor de la llibreria (*DS1 part 2*). Sempre que es tingui un cub creat, s'enviaran als shaders de la GPU les dades del cub, les de la càmera, les textures del volum i la màscara, un llindar de saturació i un valor de mostreig . Amb totes elles, els shaders duran a terme l'algorisme del Ray Casting de volum en mode MIP i usant la tècnica *One Step* (en concret, el fragment shader). Posteriorment el *draw()* farà el dibuixat del cub i llegirà els píxels dels dos buffers resultants del fragment shader amb la instrucció *glReadPixels()*, que els capturarà i es retornaran cap a la classe principal que ho retornara cap a MATLAB.

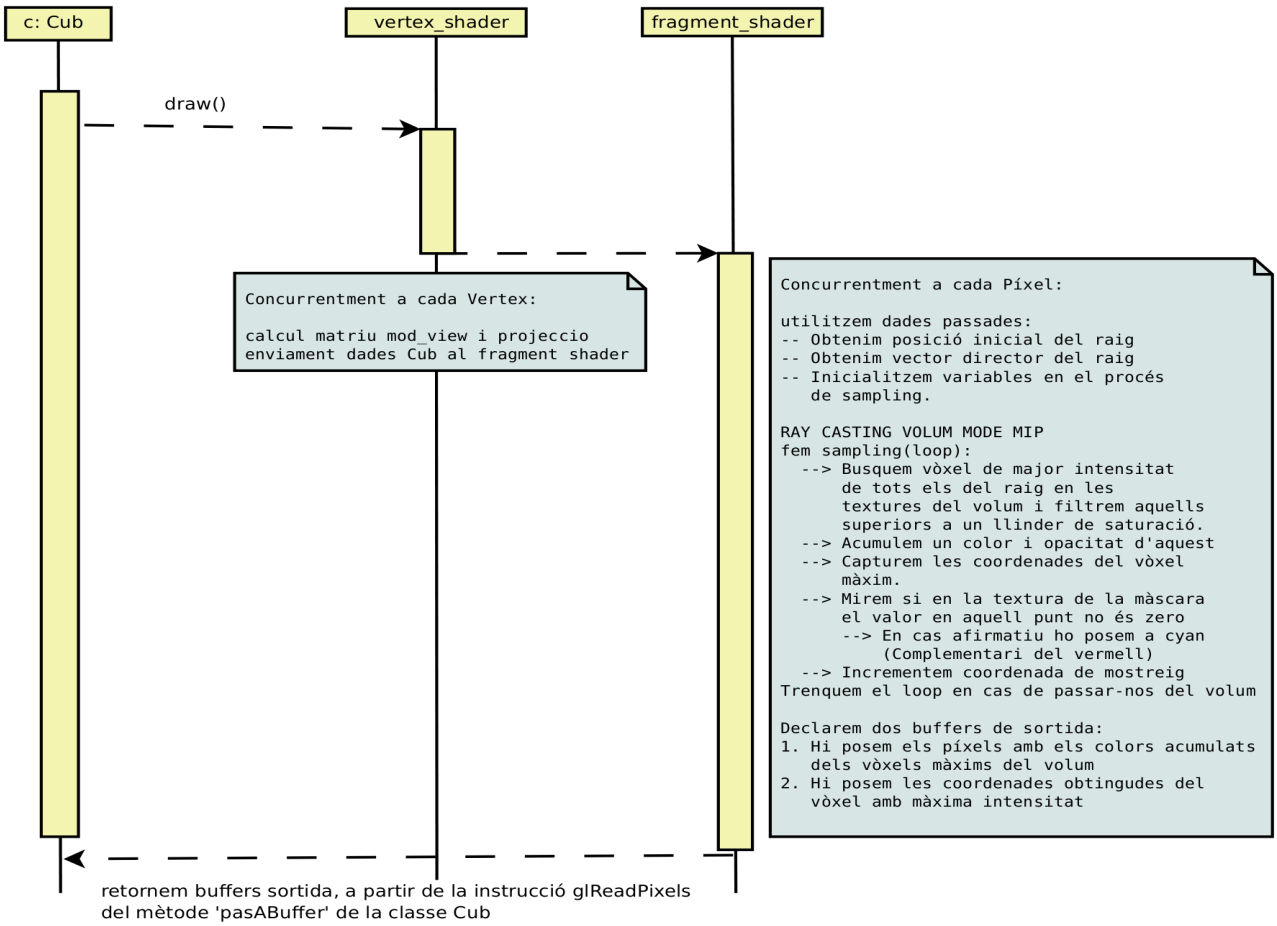


Fig. 31 – DS1 Part 4

Finalment aquesta part mostra el procés de GPU a partir de vertex shader i fragment shader. Veiem el pas de les dades i com s'utilitzen aquestes per fer l'algorisme del Ray Casting de volum en mode MIP i retornar les sortides cap a la llibreria i posteriorment, cap a MATLAB, on es mostrarà el resultat en la interfície.

4.2.2 Diagrama de seqüència 2 (DS2)

Actualitzar la visualització del volum a partir de la interacció per modificar els paràmetres de la càmera i canviar el punt de vista del volum.

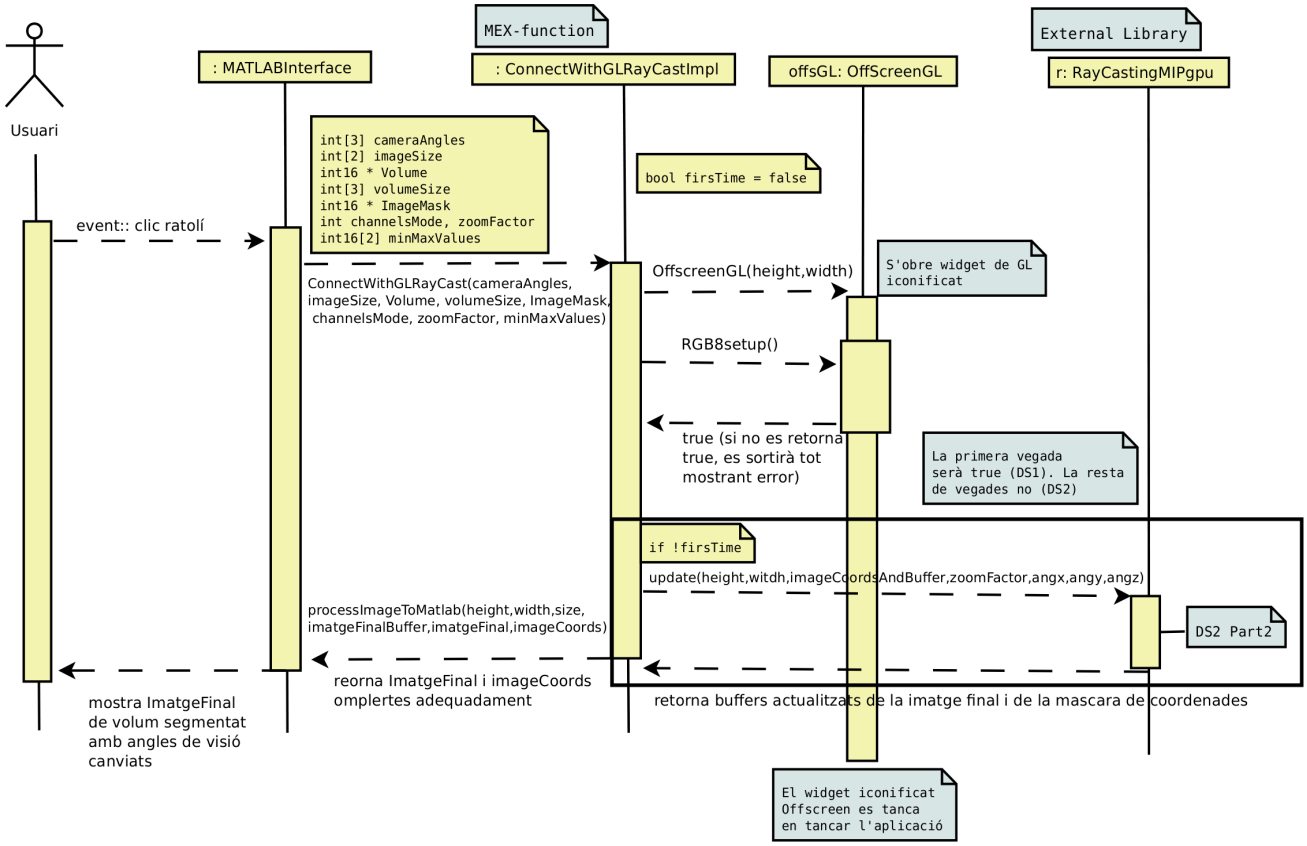


Fig. 32 – DS2 Part 1

El present diagrama de seqüència és similar al *DS1 Part 1* ***(veure Fig. 28)***, on es veu com s'estableix la connexió entre MATLAB i OpenGL. La diferència ara és que l'event d'activació és un clic de ratolí i que el booleà té valor fals, pel que ja no es la primera vegada que es crida i només voldrem actualitzar (*update()*) la imatge final del volum perquè es visualitzi des d'un punt de vista diferent. Igual que en el *DS1* es retorna de la GPU el nou buffer de píxels de la imatge i la màscara de les coordenades.

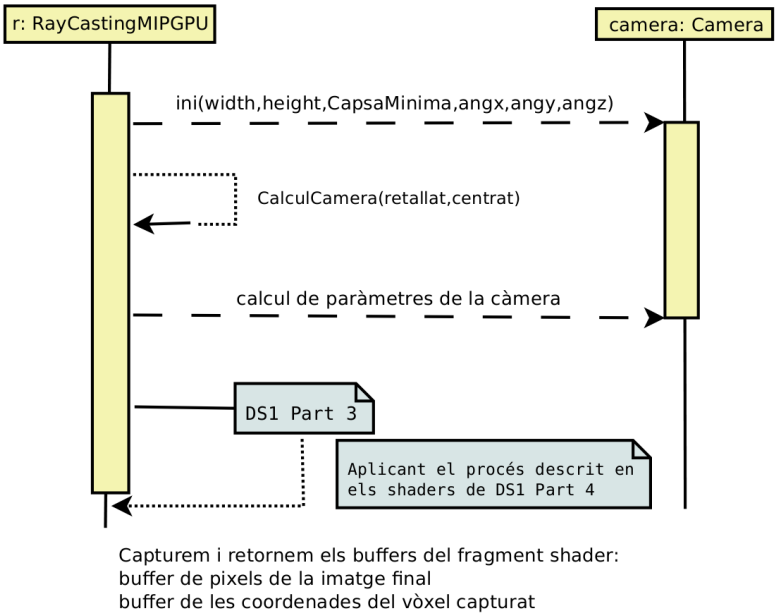


Fig. 33 – DS2 Part 2

En la segona part del diagrama de seqüència 2 s'observa com en cridar al mètode d'actualització, ens estalviem tota la part d'inicialització i càrrega del volum mostrada en el *DS1 part 2* ***(veure Fig. 29)*** i que només reinicialitzem i recalculem la càmera amb els nous angles de visió passats. Després, cridem de nou al mètode *draw()* perquè torni a fer el procés de redibuixat, pas de dades a GPU i es torni a emprar l'algorisme de Ray Casting de volum en modi MIP al fragment shader amb la càmera actualitzada (aquest procés és exactament el mateix que es descriu i mostra en el *DS1 part 3*). Finalment es retornen també els buffers de la imatge i les coordenades amb el mateix procediment que el descrit en el *DS1 Part 3* ***(veure Fig. 30).***

La part del procés de la GPU en el fragment shader i el vertex shader serà la mateixa que es mostra i descriu en el *DS1 Part 4* però amb les dades de la càmera actualitzades ***(veure Fig. 31).***

4.2.3 Diagrama de seqüència 3 (DS3)

Actualitzar la visualització de volum modificant el contrast de la imatge final.

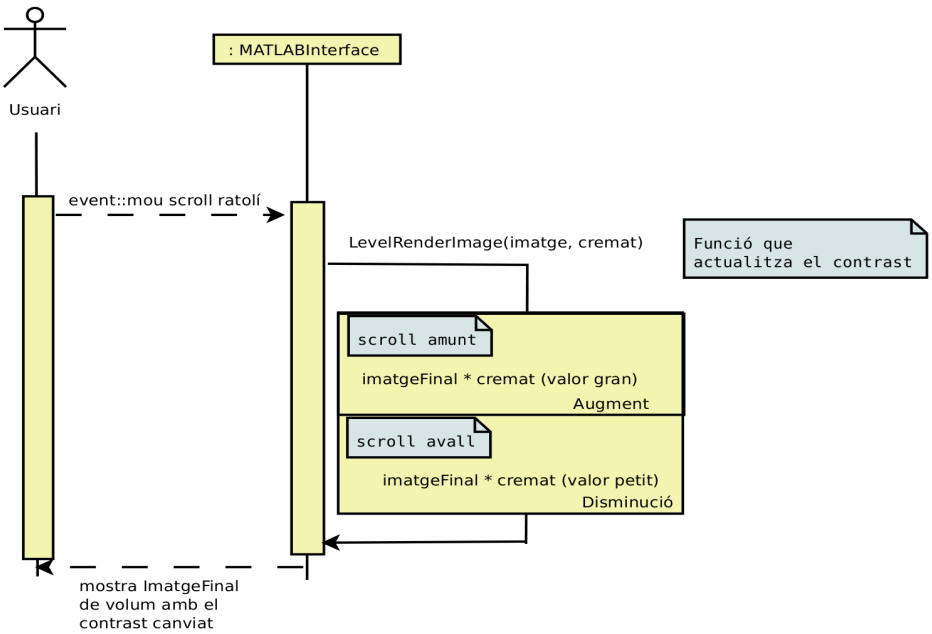


Fig. 34 – DS3

En el tercer i últim diagrama de seqüència veiem com es duu a terme el procés de modificació del contrast de la imatge final del volum mostrada en la interfície de MATLAB. L'usuari, a partir de l'event de moure la rodeta (scroll) del ratolí activarà aquest procés. El contrast es disminuirà o augmentarà segons el moviment de l'scroll.

5. Implementació

En aquest capítol es detallaran aspectes referits a la implementació i desenvolupament del projecte, així com explicacions sobre el funcionament intern de la llibreria i de la connexió d'aquesta amb MATLAB.

Com ja s'ha indicat en capítols anteriors, el que es pretén és integrar en una aplicació software de MATLAB de visualització de volums mèdics RMI-PET segmentats, un algorisme de renderitzat extern que sigui capaç d'accelerar el procés de visualització i interacció en temps real que resultava tant lent en un principi. Pels motius exposats prèviament, la millor opció és la d'utilitzar l'algorisme de Ray Casting de volum en mode MIP i sobre la GPU. Degut a que serà implementat sobre la GPU, MATLAB necessitarà del proveïment de certes llibreries i frameworks que ofereix OpenGL.

Es recomana anar seguint el diagrama de classes i els diagrames de seqüència del capítol anterior per tenir una visió més concreta sobre les explicacions les classes o els mètodes d'aquestes que s'analitzen en el present capítol.

5.1 MATLAB: Connexió OpenGL, MEX-function i crida a la llibreria

En primer lloc és necessita establir una connexió MATLAB-OpenGL, ja que el flux principal va des de MATLAB a la funció del Ray Casting de volum sobre la GPU, que ens retorna la imatge processada amb les zones patològiques i una màscara de les coordenades dels vòxels trobats de més intensitat (MIP), per tal de mostrar-ho en una interfície de MATLAB, i poder fer la interacció posterior ***(veure Fig. 35)***. Perquè això pugui realitzar-se amb èxit, caldrà proveir MATLAB de les llibreries corresponents que permetin aquesta connexió i fer el seu linkat (procés descrit en el manual tècnic del desenvolupador de l'apèndix A.2).

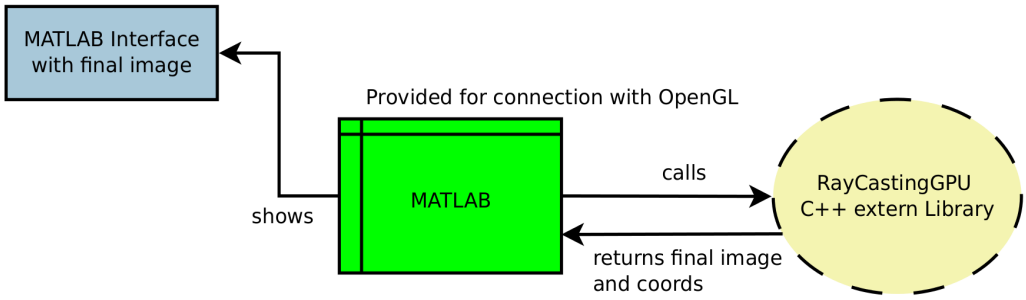


Fig. 34 – Esquema bàsic sobre el funcionament

MATLAB disposa d'un proveïdor d'interfícies per a subrutines en llenguatge C/C++ denominat 'MEX', i utilitza les funcions 'MEX-functions' per poder implementar codi en C/C++ i passar-li els paràmetres d'entrada i sortida des de les funcions de MATLAB. Una 'MEX-Function' té sempre un mètode principal 'mexFunction()' amb una capçalera comuna:

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
```

On 'nlhs' és el nombre d'entrades de la funció, *plhs[] un punter a tot el conjunt d'entrades, 'nrhs' el nombre de sortides i '*prhs[]' un punter al conjunt de sortides.

Per tant, amb tot això, el primer pas serà realitzar la connexió de MATLAB i OpenGL per tal de poder programar sobre la GPU, retornar el buffer de píxels del nostre volum mèdic, passar-lo, i visualitzar-lo en una interfície de MATLAB en RGB. Per dur-ho a terme, es necessari abans obrir un context de GL mitjançant la creació d'un widget d'OpenGL que dibuixi i mostri el resultat per després poder capturar-ne el buffer. Això és necessari ja que el context GL necessita disposar d'un widget obert per poder obtenir el buffer de píxels del volum processat. Per efectuar aquest pas, s'ha implementat un *header* de C++ en MATLAB (*OffscreenGL.h)* que implementa una classe que realitza la inicialització de paràmetres útils per connectar i utilitzar OpenGL, la creació dels framebuffers, i la creació del widget necessari on pintar el resultat (posteriorment capturat en el buffer). El widget creat utilitza la llibreria GLUT d'OpenGL i per a que no aparegui conjuntament amb la interfície de MATLAB, es manté iconificat a fi que passi inadvertit per l'usuari de l'aplicació.

El següent pas, un cop establerta la connexió amb GL, és crear una llibreria externa

“RayCasting” en C++, que permeti realitzar la crida a les funciones necessàries per processar els vòxels del volum i la màscara, connectar amb els shaders de la GPU, i realitzar els procediments necessaris per obtenir i llegir la imatge desitjada i la màscara de les coordenades utilitzant l'algorisme de traçat de rajos en volum i mode MIP (la llibreria ha de ser externa, ja que les 'MEX-functions' no disposen de tot el necessari per a poder implementar el requerit). Aquesta llibreria permetrà el pas des de MATLAB del volum mèdic, la màscara de segmentació i altres paràmetres importants com la mida del volum, característiqueses de la càmera i la mida de la imatge final. Com a retorn s'obtindran dos buffers: el la imatge del volum mèdic i el de la màscara de les coordenades, sempre i quan s'hagi efectuat prèviament la connexió amb OpenGL i l'obertura i iconificació del widget de Glut. Un cop realitzat amb èxit tot aquest procediment, ja es tindrà tot el necessari per poder dur a terme la integració amb l'aplicació final.

A continuació es detalla de forma esquemàtica com es duu a terme el procés de connexió OpenGL-MATLAB i la crida de la llibreria amb el pas de paràmetres, seguint el flux dels diagrames de seqüència definits en el capítol anterior (DS 1 i DS 2)

- Primerament, tenim una funció MATLAB *ConnectWithGLRayCast.m* que inicialitza '*Glut*' i conté una crida a la MEX-function base *ConnectWithGLRayCastImpl.cpp* que realitza tots els passos descrits. Aquesta funció és la que serà integrada posteriorment a l'aplicació final. Exemple de la funció:

  [imatgeFinal,imageCoords] = ConnectWithGLRayCastImpl(CameraAngles, imageSize, Volume, volumeSize, MaskVolume, channelsMode, zoomFactor, minMaxValues)

- La MEX-function *ConnectWithGLRayCastImpl.cpp* rep per paràmetre les entrades i sortides necesssàries per a les crides de les funcions de la llibreria externa, i, un cop fet tot el procés intern, MATLAB podrà agafar les mateixes sortides ja plenes amb els valors adequats.

  - Internament, la MEX-function importa primerament totes les llibreries necessàries, entre elles la *RayCastingMIPgpu*.

  - El seu mètode principal (la capçalera del qual és com la mostrada abans):

- Captura totes les entrades en variables separades (del punter del conjunt d'entrades) i realitza el mateix per a les sortides.

- Crida el constructor del header implementat *OffscreenGL.h* que connecta amb el context OpenGL i obre el widget "*Glut*" iconificat per poder capturar el buffer de sortida un cop fet tot el renderitzat.

- Seguidament crida una mètode de *OffscreenGL.h* que inicialitza i declara els framebuffers on es posaran els píxels a dibuixar en el widget que posteriorment seran capturats.

- Si el pas anterior es realitza correctament, aleshores es crida el mètode intern "*renderImageGL()*" amb tots els paràmetres d'entrada i sortida necessaris.

- Dins el mètode *"renderImageGL()"* és on es duu a terme la crida de les funcions de llibreria interna *RayCastingMIPgpu,* que realitzarà el renderitzat de volum en GPU. A partir d'un booleà com a variable global, es controla la primera vegada que es crida la funció de la resta de vegades. D'aquesta manera es diferencia:

    * La primera vegada, on s'ha de processar el volum i la seva màscara de segmentació, inicialitzar paràmetres del GL, i posteriorment (amb el Ray Casting mode MIP) visualitzar-lo amb els buffers obtinguts de la GPU.

    * La resta de vegades, on només voldrem dur a terme una actualització dels paràmetres de la càmera per veure el volum des de diferents punts de vista, i redibuixar-lo obtenint els nous buffers. No caldrà processar els vòxels del volum i la màscara de nou i ens estalviarem molt temps.

- Un cop dut a terme tot aquest procés, es capturen en un doble punter el dos buffers de sortida obtinguts del retorn de la llibreria (imatge final i màscara de coordenades dels vòxels dibuixats) i es crida el mètode *processImageToMatlab()* que s'encarregarà d'omplir correctament les dues matrius de sortida "*imatgeFinal*" i "i*mageCoords*", declarades inicialment. Aquestes, seran recuperades per MATLAB amb els valors dels píxels obtinguts i finalment, es mostrarà la imatge final amb la instrucció MATLAB pertinent utilitzant el resultat obtingut en la matriu de sortida.

** El widget de "Glut" generat en "OffscreenGL.h" romandrà sempre obert per tal de poder pintar el contingut de la sortida i capturar els buffers de sortida, però la seva presència serà inapreciable ja que estarà iconificat fins a apagar l'aplicació.*

A continuació es mostra un diagrama de flux ***(veure Fig.36)*** de tot el que s'ha explicat anteriorment, per fer més fàcil l'enteniment del funcionament:

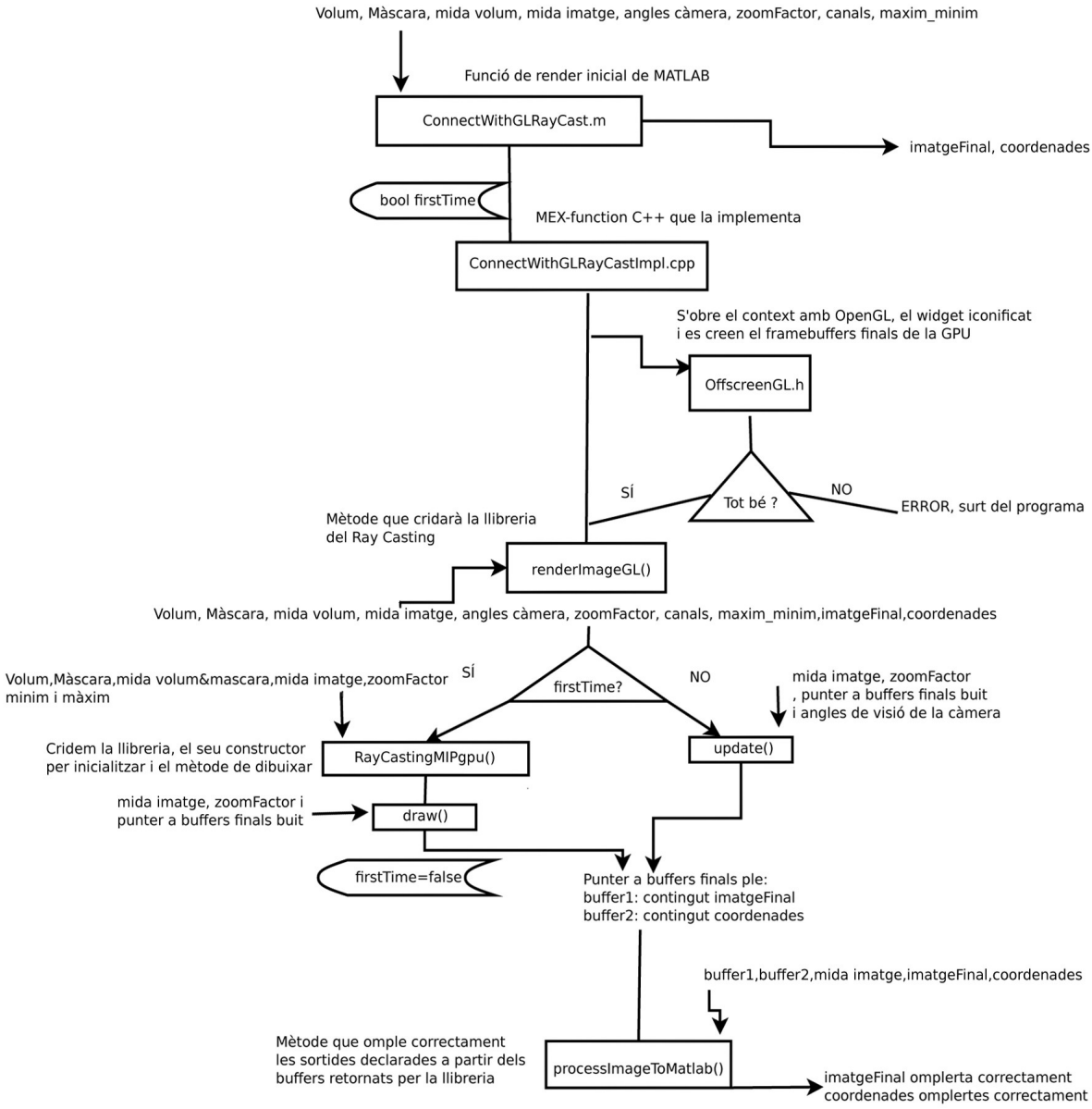


Fig.36 – Diagrama de flux connexió MATLAB-OpenGL i crida a la llibreria

5.2 Paràmetres d'entrada i sortida de la funció principal

Un cop explicat el funcionament anterior, es passarà a explicar els paràmetres d'entrada i sortida que es passen i s'obtenen de la funció MATLAB del renderitzat en Ray Casting de volum. Tanmateix, es detallarà la necessitat i utilitat d'aquests dins la llibreria externa.

5.2.1 Entrades:

La funció de MATLAB (*ConnectWithGLRayCastImpl*) del renderitzat en Ray Casting de volum en mode MIP sobre la GPU i connexió amb OpenGL, conté els següents paràmetres d'entrada:

- ***CameraAngles*:**
  És un vector de tres enters que conté els angles de visió (x ,y i z) de la càmera utilitzada dins la llibreria. Serveixen per canviar de punt de vista la visió de la imatge final del nostre volum segmentat quan es pateix una interacció amb aquest.

  Dins la llibreria es passen per separat i s'utilitzen especialment en la funció *update()* per recalcular la càmera i altres components i així actualitzar la visió del volum i redibuixar-lo amb els nous angles de visió.

- ***imageSize:***
  És un vector de dos enters que conté la mida (ample i alt) de la imatge 2D final on es plasmarà el renderitzat del volum (el widget). Per defecte, en la interfície de MATLAB de l'aplicació final la mida és sempre de 400x400.

  L'ample i alt són essencials per determinar la mida final dels buffers de retorn, la mida del viewport del widget que apareixerà iconificat i la quantitat de píxels que s'hauran de retornar en el buffer.

- ***Volume:***

  És el volum mèdic RMI que cal processar, o millor dit, el conjunt d'intensitats del món de vòxels d'un volum determinat.

  Els volums estan emmagatzemats en arxius ".mat" adients per guardar matrius en MATLAB. És fàcil carregar-los amb instruccions MATLAB i passar-los com a un punter a un tipus. El tipus de dades internes (les intensitats dels vòxels) són del tipus *int16*. Aquest fet, fa que en passar-ho al llenguatge C++, el tipus *int16* passi a *short int*, ja que és el seu equivalent segons la documentació dels "*data types*" de C++ (un enter normal ocuparia 32 bits , mentre que un *short int* és de 16 bits).

  Dins la llibreria, el pas del volum només es realitza la primera vegada per poder carregar el volum, obrir-lo, processar els seus vòxels, guardar-los, crear la textura i passar-la cap a la GPU (els shaders) on se li aplicarà l'algorisme de Ray casting de volum en mode MIP . La resta de vegades el volum ja estarà carregat i passat a GPU i no caldrà passar-lo més, pel que només caldrà fer un 'update' canviat els angles de visió.

- ***volumeSize:***

  És un vector de tres enters que representa tant la mida del volum passat per paràmetre com la de la seva màscara de segmentació.

  Tècnicament, representa la mida (alçada, amplada i profunditat) del món de vòxels a processar. Dins la llibreria es passen per separat i són utilitzats en funcions que necessiten recórrer els vòxels del volum i la màscara per processar-los (les funcions de càrrega i obertura del volum) i després crear-ne les textures.

- ***MaskVolume:***

  És la màscara de segmentació del volum determinat passat per paràmetre.

  Cada volum té la seva pròpia màscara de segmentació amb les mateixes dimensions i tipus de dades que ell, per tant, el tipus serà també un punter de *int16* en MATLAB i

un punter de *short int* en C++. També s'emmagatzemen en arxius '.mat' de MATLAB. Són binàries i cada valor d'aquesta correspon al vòxel equivalent en la mateixa posició del volum original. Els uns representarien una zona patològica a ser pintada de color vermell.

El seu protagonisme dins la llibreria és exactament el mateix que el del volum: només es passarà una sola vegada per poder-la carregar (la resta ja no caldrà), processar els seus valors, crear la textura i passar-la als shaders de la GPU on aplicant l'algorisme Ray Casting de volum en mode MIP al volum original, es pintaran en vermell aquells vòxels que coincideixin amb un valor diferent a zero en la seva màscara de segmentació.

- ***channelsMode:***
  És l'enter que determina el número de canals que contindrà la imatge final. Tres canals indicarien que es vol RGB i quatre que es vol RGBA. Per defecte es declara de 3 ja que la imatge resultant la volem RGB, sense tenir en compte el canal alpha.

  En la llibreria es passa un doble punter a dos buffers de *floats* de mida (400x400x3), que seran els dos buffers finals a omplir i retornar un cop omplerts. Primerament, es passa aquest doble punter buit, i, un cop s'ha dibuixat la imatge en el widget i es llegeixen els píxels dels dos buffers retornats per la GPU, s'omple cada un dels dos buffers amb els valors *float* RGB de la imatge i les coordenades corresponents, i es retorna. Finalment, s'agafa aquest punter amb els dos buffers plens i en un mètode intern es duu a terme l'emplenat de les matrius finals de sortida (*imatgeFinal* i *coordenades*), de tal manera que els canals RGB quedin ordenats.

- ***zoomFactor*:**
  És un enter que representa el factor de zoom de la imatge final i com a tal determinarà també la mida del buffer final. Per defecte es posa a 1, i per tant és negligible en el resultat final. Si es volgués augmentar la imatge final a un factor de zoom de 2, aquesta apareixeria més gran. Dins la llibreria només té utilitat a l'hora de determinar la mida dels buffers finals i el viewport.

- ***minMaxValues*:**

  És un vector de dos enters de 16 bits ('short int' en C++) que representen el valor màxim del món de vòxels del volum i el valor mínim. Aquests paràmetres s'utilitzen en la funció d'obertura i processat de vòxels del volum per escalar els valors dels vòxels de manera que no hi hagi pèrdues significatives ni qualitatives en la imatge final degut a la conversió de tipus, ja que, inicialment, es té *short int* (de 16 bits) i la imatge resultant ha de ser de *doubles*, pel que la gamma de colors quedaria reduïda i amb pèrdues en l'escala de grisos degut al canvi de tipus. D'aquesta manera es filtren valors per sota d'un llindar i la resta s'escalen respecte el màxim valor, obtenint una gamma de grisos més variada i amb un escalat molt menys brusc que sense la filtració.

5.2.2 Sortides:

Los sortides de la funció del renderitzat de MATLAB són dues:

- ***imatgeFinal:***

  És una matriu de *doubles* de dimensions: 400x400x3 (alt de la imatge, ample, i tres canals de representació de cada píxel: RGB). Aquesta, contindrà els píxels obtinguts de la GPU un cop aplicat l'algorisme del Ray Casting de volum en mode MIP, per tant, serà el resultat de la nostra imatge final del volum a plasmar en la interfície de MATLAB .

- ***imageCoords:***

  És l'altra matriu de sortida, també de 400x400x3, que contindrà per a cada píxel les coordenades (x, y i z) on es trobava el vòxel de major intensitat de tots els travessats pel mateix raig en l'algorisme del Ray Casting de volum. Com s'ha explicat en l'*apartat 2.3*, ens interessa per a la interacció i actualització del volum.

Dins la llibreria externa, es passa per paràmetre, tant al mètode de dibuixar com al

d'actualitzar, un doble punter a dos buffers de *floats* de mida 400x400x3 que capturaran els dos buffers de sortida del fragment shader de la GPU (el dels píxels de la imatge final i el dels valors de les coordenades dels vòxels capturats). Primerament, es passa el doble punter amb els dos buffers buits i, un cop ple amb els valors de la imatge i les coordenades corresponents capturats del fragment shader amb una instrucció de GL, es retorna de la funció . D'aquesta manera, s'obtenen com a retorn els dos buffers passats per paràmetre ja plens amb els valors necessaris per omplir les matrius de sortida de la funció de MATLAB.

Finalment, per passar els valors del buffers retornats a la matrius de sortida corresponents, es crida un mètode intern *processImageToMatlab()* que processa els píxels i s'encarrega d'omplir les matrius de sortida: i*matgeFinal* i *imageCoords* correctament. En el procés d'emplenat de les matrius de sortida amb els valors dels buffers, cal tenir present els següents aspectes :

- S'emmagatzema successivament cada valor R-G-B de cada píxel un darrere l'altre (R1,G1,B1,R2,G2,B2...), on el número indica el píxel. Això fa que en el procés d'emplenat de les matrius finals, calgui, per a cada píxel, separar el valor d'intensitat del canal vermell (R), el verd (G) i el blau(B) i posar-los en la posició de la submatriu del seu canal corresponent (valor d'intensitat de vermell del píxel 1, a la posició 1 de la submatriu 1; valor del verd; a la posició 1 de la sumbatriu 2, i el mateix pel blau en la submatriu 3. La concatenació de les submatrius 1,2 i 3 de 400x400 cada una, dóna lloc a la matriu RGB de la imatge). Per la matriu de les coordenades no hi ha el problema dels canals (ja que no es mostrarà com a imatge), però també s'hauran de separar de la mateixa manera els valors: en la submatriu 1 els valors x's de les coordenades, en la 2 els de les y's i, en la 3, els de les z's.
- En els buffers obtinguts els píxels estan (per defecte) ordenats d'últim a primer (el valor R-G-B de la imatge o el valor de la coordenada x-y-z del vòxel del primer píxel es troba en última posició del buffer i el de l'últim en la primera). Per tant, en el procés d'omplir les matrius cal recórrer els buffers al revés.
- Les matrius de MATLAB s'omplen per columnes i no per files i els valors dels buffers estan emmagatzemats en un ordre propici a les files. En el procés d'emplenat cal modificar els índexs adequadament per tal que puguin fer l'emplenat idoni.

En la següent figura s'il·lustra el procés de retorn i emplenat de les sortides anteriorment explicats ***(veure Fig.37)***:

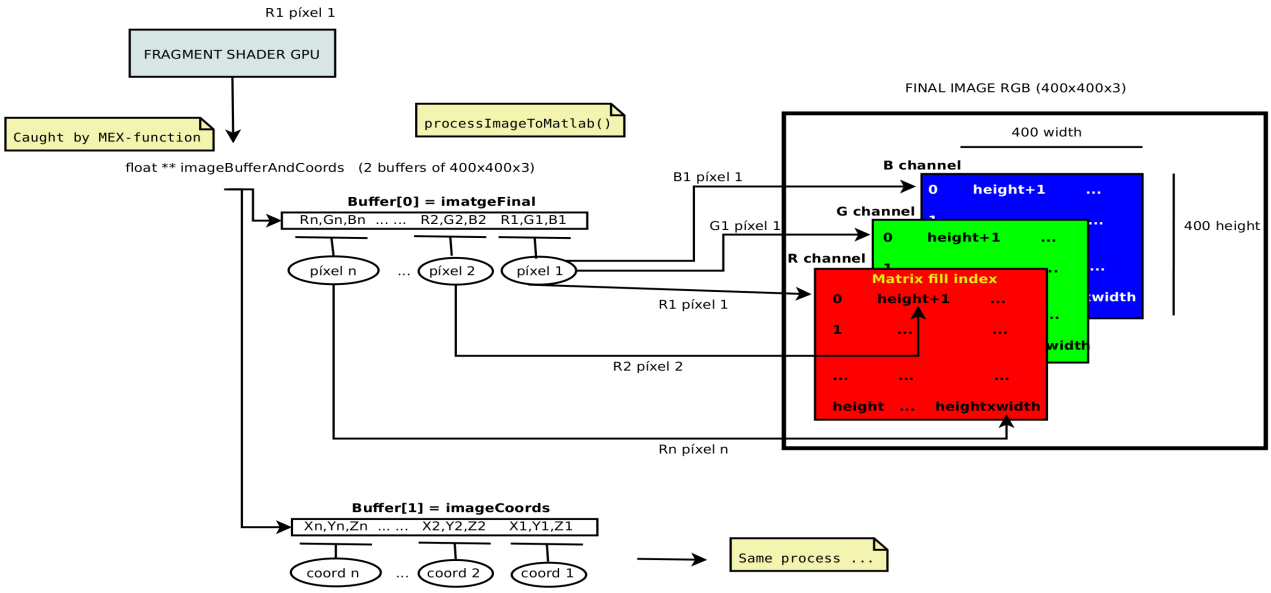


Fig.37 – Esquema del procés d'emplenat de matrius finals a partir del buffer de retorn

** Els buffers es capturen i llegeixen en RGB i tipus float (tipus de dada més còmoda per a les sortides del fragment shader). La seva mida serà la de la matriu de la imatge final (alt x ample x canals), ja que cada píxel, representa un valor (nivell de gris) en cada unitat mínima de la imatge a plasmar.*

** Per a poder fer la lectura i captura dels buffers de píxels de la GPU (fragment shader), es requereix la presència d'un widget de GL obert que pinti els píxels de sortida (en aquest, cas la finestra de "Glut" que es manté iconificada).*

5.3 Llibreria externa:

La llibreria externa implementada en llenguatge C++ conté el conjunt de classes i headers necessaris per a poder dur a terme la visualització del volum mitjançant el Ray Casting en

mode MIP. Està proveïda de les llibreries OpenGL necessàries per programar i interactuar amb la GPU, i d'una sèrie de headers que la proveeixen de tipus de dades que seran necessaris en el seu codi intern: matrius 4x4 (mat4), vectors de 3 elements (vec3), vectors de 4 (vec4) i altres estructures i tipus. En el diagrama de classes i els diagrames de seqüència del capítol anterior, es pot apreciar els elements pels quals està formada i la comunicació que hi ha entre aquests.

Si s'han seguit els subcapítols anteriors del present capítol (*5.1 i 5.2*), sabrem com es realitza la crida a aquesta llibreria, la seva necessitat i el que aquesta retorna (el punter als dos buffers resultants: el corresponent a la imatge final i el de la màscara de coordenades). Cal remarcar també, que abans que aquesta sigui cridada caldrà que estigui obert el context de GL, s'hagin creat els framebuffers corresponents i es mantingui obert i ocult (per no molestar l'usuari) un widget on poder pintar el resultat (tot això ho subminsitra la *OffscreenGL*, detallada en el *5.1 i el capítol 4*).

Un cop recordat tot aquest procés, cal fer especial menció en què el flux de la llibreria es pot dividir en dues parts diferenciades: El de la primera vegada que es crida, on es duran a terme uns determinats passos tals com la inicialització de paràmetres o la càrrega del volum, i el de la resta de vegades que s'ha cridat, on ja només es voldrà actualitzar els paràmetres de la càmera per veure el volum ja carregat des de diferents punts de vista. Aquesta diferència serveix per disminuir el cost temporal del renderitzat quan l'usuari només vol actualitzar la visió del volum. Així, la primera vegada que es cridi la funció del renderitzat en Ray Casting MIP GPU, caldrà carregar el volum i la màscara, compilar i linkar els shaders i inicilaitzar el GL. Per altra banda, la resta de vegades, on l'usuari voldrà interactuar amb el volum i ja l'haurà carregat, només caldrà canviar els paràmetres de la càmera i recalcular la sortida. La resta de processos duts a terme la primera vegada de la crida, ja no seran necessaris, i en conclusió, el cost temporal serà inferior al de la primera vegada.

Tant si és la primera vegada com si no, sempre caldrà accedir als shaders de la GPU (vertex shader i fragment shader) per dur a terme el pas de paràmetres i calcular o recalcular el Ray Casting de volum en mode MIP amb el volum i la màscara passats. Finalment es llegirà

i retornarà el resultat del fragment shader en forma de dos buffers a ser processats posteriorment en MATLAB i mostrar-ho en la interfície (tal com es descriu en *l'apartat 5.2.2*).

A continuació es detallarà el flux intern de la llibreria durant la primera vegada que es crida la funció del render i el de la resta de vegades. El procés que es duu a terme en els shaders de la GPU, i per tant, la implementació i desenvolupament del Ray Casting de volum en mode MIP realitzada al fragment shader, està detallada en el diagrama de seqüència 1 part 4 (*DS1 Part4*) del capítol anterior referent al disseny, pel que no es repetirà de nou en aquest capítol per evitar redundàncies ***(veure Fig. 31).***

5.3.1 Primera vegada de crida (Extensió de DS1):

El control d'un booleà dins la MEX-function "ConnectWithGLRayCastImpl", és el que permetrà establir quan ens trobem en la primera vegada de crida a la llibreria i quan en la resta. Cal que aquesta MEX-function importi la nostra llibreria amb l'include de la classe principal (*RayCastingMIPgpu*) i que creï un objecte d'aquesta.

Un cop creat l'objecte, si ens trobem en la primera vegada de la crida, caldrà dur a terme dos passos:

1. Primerament, cridem el constructor de l'objecte de la llibreria creada. El *RayCastingMIPgpu()* és el constructor de la classe principal (que té el mateix nom). Rep per paràmetres: el món de vòxels del volum, la màscara de segmentació associada, l'alt, ample i profunditat del món de vòxels del volum, l'alt i ample de la imatge final, un punter al màxim i mínim valor del món de vòxels per fer l'escalat i el factor de zoom. Amb tots aquests paràmetres, realitza la crida de tres funcions:

   1.1 En primer lloc, crida a *initializeGL()*. Aquesta, realitza una sèrie d'inicialitzacions i activacions bàsiques de GL per poder dibuixar en el widget. Posteriorment inicialitza la càmera amb el mètode *ini()* d'aquesta on s'inicialitzaran i calcularan els seus paràmetres: el VRP, els angles de visió (x,y,z) inicials...etc. Finalment mitjançant el mètode *CalculCamera()* s'indicarà amb dos booleans per paràmetre

que es vol una escena sense retallat i centrada (la que desitgem), i es faran els càlculs de la càmera i l'escena necessaris per establir aquestes dues opcions: càlcul de la capsa mínima contenidora de l'escena, càlculs de la matriu model-view, de la matriu de projeccció, la finestra...etc.

1.2 Seguidament es crida a *defineViewport()* que simplement defineix la mida del viewport de la nostra escena, a partir dels paràmetres passats d'alt, ample i factor de zoom de la imatge final.

1.3 Un cop fets els dos processos anteriors, es passa a cridar el mètode *cargarVolumen()*. Aquest mètode rep com a paràmetres: el món de vòxels del volum, la màscara de segmentació, les dimensions del món de vòxels i els valors mínim i màxim d'aquest. En primer terme, crea un objecte *Volumen* i li passa al seu constructor l'ample, alt i profunditat del món de vòxels. Un cop creat l'objecte:

1.3.1 Es va al mètode *abrirVolumen()* de *Volumen* i hi passem el punter del món de vòxels del volum carregat en l'aplicació i els valors màxim i mínim d'aquest. En ell, es declara un buffer en memòra anomenat '*voxels*' on s'hi aniran posant els vòxels que es van processant en l'ordre pertinent i sotmesos a un escalat entre el valor màxim, el mínim i un filtrat a partir d'un llindar, per tal de no perdre qualitat en la imatge final degut a la conversió del tipus de dades (el que es descriu en *l'apartat 5.2*). Els vòxels passaran a ser *floats* que és com els volem passar a GPU i com volem que aquesta ens els torni.

1.3.2 Després s'accedeix al *setMask()* al qual li passem el punter a les dades de la màscara de segmentació associada al volum i carregada conjuntament. Anàlogament al processat de vòxels anterior, omplim amb els valors de la màscara un buffer en memòria anomenat '*maskpoints*' aprofitant que la mida d'aquest serà la mateixa que la del món de vòxels del volum. En aquest cas, no caldrà cap escalat ja que la màscara és binària i només té valors de 0 o 1.

1.3.3 Duts a terme el processat de vòxels del volum i de les dades de la màscara en els dos passos anteriors, el següent pas és anar a *addVolumen()* que fixarà l'ample, alt i profunditat del volum per després passar-los al constructor del *Cub*,

i, en última instància, accedeix al mètode *creaTextura3D()*. A causa que s'utilitza la tècnica de renderitzat en GPU d'un sol pas (*One Step, veure apartat 2.1.6.1*), la GPU utilitza informació del cub per dur a terme el Ray Casting de volum i calcular el raig. Aleshores, cal dibuixar les *frontface* del cub per trobar el punt d'origen del raig. Per tal que el resultat final no sigui la visualització de les *frontface* del cub, caldrà transformar el conjunt de vòxels guardats en el buffer '*voxels*' en textures3D de GL. D'aquesta manera, s'afegiran sobre el cub (de les mateixes dimensions que el món de vòxels) les textures3D del volum i la màscara (després de passar pel mode MIP en l'algorisme de Ray Casting) donant com a resultat la imatge del volum segmentat desitjat (tot i que en realitat es una textura3D sobre un cub de les mateixes mides).

De tot el que s'ha descrit anteriorment, se n'encarrega el mètode *creaTextura3D()*, que recorre els buffers '*voxel*' i '*maskpoints*' omplerts en els passos *1.3.1 i 1.3.2* i els va posant en uns altres dos buffers: '*volumeData*' i '*maskData*', que tindran format de GL. Finalment, crea dues textures 3D per ser passades cap a la GPU: la del volum '*texturaVolum*' (que contindrà el contingut del buffer de '*volumeData*') i la de la màscara '*texturaMask*' (que contindrà el contingut del buffer de '*maskData*'). Ambdues textures seran de tipus RGB i de les mateixes dimensions que el món de vòxels.

1.3.4 Amb les textures del volum i de la màscara creades, el següent i últim pas dins el *cargarVolumen()* és cridar el mètode *rcGPU()*. En aquest mètode es fixen els valors (a un número arbitrari que es podrà canviar internament) dels paràmetres de la saturació i el valor de mostreig (sampling). Aquestes dades es passaran després cap a la GPU i s'utilitzaran en l'algorisme de Ray Casting del fragment shader. La saturació serà un llindar que filtrarà certs valors del vòxel que no siguin superiors a ella, donant una imatge més o menys saturada. El valor de mostreig és l'increment del raig, la mida del seu pas (com més petit sigui el pas, més detallades es veuen les imatges i com més gran, menys detallades). Un cop fixats aquests valors, es cridarà a *startRayCasting()* que:

- Crea un *Cub* amb l'ample, alt i profunditat que tenia el volum.

- Realitza el *make()* de l'objecte *Cub* creat anteriorment. En el *make()* es cridarà una sèrie de mètodes recursius que s'encarregaran d'anar calculant els vèrtexs del cub a generar i a partir d'aquí, s'emmagatzemaran els punts, colors i coordenades de textures d'aquest que després es passaran a la GPU per aplicar el One Step.

- Es crida a *initShadersGPU()* que passa al mètode *InitShader()* el *paths* de la ubicació on es troben tant el vertex shader com el fragment shader (ambdós es troben en la carpeta dels arxius de MATLAB). Amb això, *InitShader()* crearà un programa GL que contindrà els shaders de la GPU. Aquests, es buscaran mitjançant el *path* subministrat i s'accedirà a ells. Posteriorment es compilaran i es linkaran. Si els processos de compilat i linkat són correctes, el programa estarà llest per usar-se i es retornarà un enter ('*program*') que identificarà aquest programa per després poder-lo usar com a paràmetre per passar-li les dades que els shaders necessitaran.

2. En segon lloc, es crida el mètode *draw()* del mateix objecte d'abans. Aquest mètode té com a paràmetres l'ample, alt i factor de zoom de la imatge final i un punter a dos buffers buits de floats amb la mida ja definida des de la MEX-function de MATLAB (400x400x3). Aquest mateix buffer és el que es retornarà ple amb les sortides obtingudes de la lectura del fragment shader de la GPU, un cop aplicat el Ray Casting de volum. Abans de tot aquest procés:

2.1 Inicialment, es comprova que hi hagi un objecte de *Cub* creat per tal de poder dur a terme correctament el procés a la GPU. En cas afirmatiu es criden tres mètodes que s'encarreguen de passar les dades i paràmetres necessaris cap als shaders de la GPU: el *toGPU() del cub*, que li passarà els punts, els colors i les coordenades de les textures del cub emmagatzemats anteriorment per poder aplicar el *One Step*, el *toGPU() de la càmera,* que li passarà els paràmetres de la càmera necessaris per efectuar la visualització (matrius model-view i projecció) i per calcular el raig (vector director), i el *dadesRCtoGPU()* que passarà la saturació i el valor de mostreig fixats anteriorment i les textures 3D creades dels vòxels del

volum i les dades de la seva màscara de segmentació, per poder aplicar el mode MIP en el Ray Casting.

2.2 Un cop passades totes les dades cap als shaders, el següent i últim pas és accedir al mètode *pasABuffer()* del cub. Aquest mètode rep per paràmetre les dimensions de la imatge final i el punter als dos buffers buits de mida definida referit anteriorment. El *pasABuffer()*, primerament realitza el *draw()* del cub que dibuixarà el cub al widget de GL i després hi aplicarà les textures del volum i la màscara amb l'algorisme del Ray Casting en mode MIP aplicat (*tècnica "One Step", veure apartat 2.1.6.1*). Un cop fet el *draw()* del cub, i havent dibuixat el volum segmentat a partir del Ray Casting en mode MIP en el widget (que estarà iconificat), es capturen en els buffers buits passats per paràmetre els dos buffers de sortida del fragmen shader. La instrucció *glReadPixels()* permet fer aquesta captura i posar en el '*buffer[0]*' el primer buffer de sortida del fragment shader que conté els píxels RGB de la imatge final del volum segmentat utilitzant el Ray Casting mode MIP. En el '*buffer[1]*' s'hi posa el segon buffer de sortida declarat en el fragment shader que dóna les cordenades (x,y,z) dels vòxels del volum capturats en el mode MIP del Ray Casting (els de més intensitat de cada raig). Finalment, ambdós buffers es retornen amb el punter passat per paràmetre (que ara contindrà els buffers plens). Aquest, es retorna també a la MEX-function de MATLAB que ha cridat el *draw()* inicial, i , un cop obtinguts els dos buffers a la MEX-function, aquesta realitzarà l'emplenat de les matrius de sortida descrit en l'apartat anterior per passar-les a MATLAB i mostrar la matriu de la imatge final del volum segmentat a la interfície.

** Els diagrames de seqüència: DS1 Part 2 i DS1 Part 3 del capítol 4, mostren clarament tot aquest procés descrit de forma més simplificada.*

5.3.2 Altres vegades de crida (Extensió de DS2):

Quan l'usuari de l'aplicació de MATLAB ja ha carregat un volum i una màscara i ha estat capaç de visualitzar-los mitjançant el Ray Casting de volum mode MIP sobre la GPU en la

interfície d'aquesta sense cap tipus de problema, el més probable és que aquest vulgui interactuar amb la interfície contenidora de la imatge del volum per tal de poder-lo veure des d'un altre punt de vista. Si l'usuari desitja realitzar aquest refresc de la imatge, a partir de l'event d'un clic de ratolí podrà dur-ho a terme i s'activarà el procés per a actualitzar la visió del volum amb uns paràmetres de la càmera (angles de visió) canviats.

Com ja s'ha mencionat en el *subcapítol 5.1*, el booleà que controla la primera vegada de la resta, estarà ja posat a *false* i la MEX-Function *ConnectWithGLRayCastImpl* només haurà d'accedir al mètode *update()* de l'objecte de la llibreria *RayCastingMIPgpu* ja creat la primera vegada.

El mètode *update()* de la llibreria rep com a paràmetres l'ample, alt i factor de zoom de la imatge final, i un punter a dos buffers de mida ja definida i amb contingut buit que després es retornarà amb el contingut dels buffers capturats del fragment shader (exactament igual que el mètode *draw()* de la primera vegada). A més a més, se li passen tres angles de visió '*angx*', '*angy*' i '*angz*' que serviran per actualitzar els paràmetres de la càmera. Per tant:

- Inicialment crida l'"*ini()* de la càmera i li passa els nous angles de visió obtinguts de la interacció amb l'usuari. La càmera actualitzarà aquests paràmetres.
- Després es torna a cridar *CalculCamera()* amb els valors de centrat i sense retallat, per recalcular tots els paràmetres de l'escena i les matrius de projecció i model-view de la càmera amb els nous angles de visió.
- Finalment crida el mètode *draw()* del cub i es realitzarà tot el procés ja descrit en el subapartat anterior (*pas 2 del 5.3.1*). Aquest retornarà la imatge del volum segmentat refrescada amb el punt el vista canviat a partir dels angles de visió i la passarà de nou cap a MATLAB.

5.4 Integració amb aplicació final:

Finalment, un cop comprovat i debuggat que els passos anteriors donen els resultats esperats, que la crida a la llibreria retorna la imatge del volum i la màscara de coordenades

esperada i que la connexió amb OpenGL s'estableix de forma correcta, només cal fer l'últim pas: Integrar i cridar la funció de MATLAB que realitza tots aquests passos (*ConnectWithGLRayCast.m*) amb l'aplicació software de MATLAB esmentada.

El pas de la integració és relativament senzill si s'ha comprovat anteriorment que tot és correcte. Només cal passar a la funció els paràmetres de la nova aplicació (volum, màscara de segmentació i mides de la imatge final i el volum), que són similars i amb el mateix tipus de dades amb què s'han fet les proves de funcionament prèvies.

Per altra banda, l'única dificultat és la integració dels paràmetres de la càmera. En la llibreria externa, es disposa ja d'una classe càmera amb els seus propis paràmetres i càlculs de les matrius de visió i projecció, en canvi, la càmera utilitzada en l'aplicació conté un altre tipus de paràmetres i càlculs diferents controlats directament per MATLAB que caldria integrar d'alguna manera. Aquest darrer pas no s'ha realitzat en el present projecte.

Tot i així, per a les proves de la interacció s'ha realitzat un canvi d'angles de visió de la nostra càmera similar al que es faria amb la càmera original, l'única limitació és que el moviment realitzat per l'usuari en la interacció serà independent a l'establert, ja que els angles es canviaran independentment de la direcció de rotació que hagi fet l'usuari amb el ratolí.

6. Resultats:

En aquest capítol es mostren una sèrie de captures que il·lustren els resultats obtinguts en aquest projecte.

S'ha realitzat una comparativa tant temporal com qualitativa de l'ús del nostre algorisme del Ray Casting de volum sobre la GPU i en mode MIP, amb l'anterior algorisme ja implementat inicialment en l'aplicació de MATLAB (el Shear-Warp). Amb dites comparatives, es remarquen les millores obtingudes de disminució del cost temporal de càrrega i visualització de volums mèdics RMI-PET, amb una màscara de segmentació incorporada.

En la primera part, es mostren una sèrie de simulacions a partir d'imatges capturades des de la interfície de l'aplicació que estableixen la comparativa entre els dos algorismes (l'original en CPU i el del present projecte en GPU). A més a més, es mostra la comparativa temporal que hi ha hagut entre la utilització d'un algorisme o l'altre amb diferents volums i discernint si és la primera vegada d'ús del renderitzat o l'actualització del punt de vista del volum mitjançant la interacció, un cop ja carregat i mostrat aquest. També s'aprecia que els resultats són qualitativament semblants.

Finalment, en l'última part, s'analitza una taula comparativa entre els dos algorismes utilitzats on es veuen els temps obtinguts en cada un d'ells amb diferents valors dels paràmetres (saturació, valor de mostreig, mida del volum ...) per tal d'emfatitzar els resultats obtinguts .

Les parts en què es divideix el següent capítol són:

6.1 *Simulacions*
6.2 *Taula comparativa de temps*

6.1 Simulacions:

Les simulacions dutes a terme per validar i il·lustrar els resultats de la funcionalitat de l'algorisme Ray Casting de volum sobre la GPU i comprovar com aquest redueix el cost temporal que es pretenia optimitzar de l'algorisme implementat en l'aplicació original han estat efectuades en el següent computador i els següents requisits hardware:

- Computador: *Sony VGN-NR32M*
- Processador: *Intel Dual-Core Processor de 1.73 GHz*
- Memòria principal: *2GB/Go DDR2 SDRAM*
- Targeta gràfica: *Intel Mobile GM965/GL960*

A continuació es mostren els resultats de les simulacions amb dos volums diferents:

Simulacions amb el primer volum:

El volum utilitzat conté un món de vòxels de dimensions: 144x 144 x 213.

Les dues primeres simulacions ***(veure Fig.38 i 39)*** corresponen a la inicialització de la primera càrrega del volum i la seva màscara utilitzant la funció del render requerida (no a la interacció).

En la primera imatge ***(veure Fig.38)*** es mostra el resultat del renderitzat del volum segmentat aplicant l'algorisme Shear-Warp que és el que ja estava implementat originalment en l'aplicació MATLAB. És l'algorisme que donava molt bona qualitat de la imatge final però que tenia un alt cost temporal que calia reduir. Cal remarcar que perquè es pugui establir la comparativa temporal entre els dos algorismes, és estrictament necessari haver aplicat la màscara de segmentació al volum, que és quan dóna els elevats costos en temps.

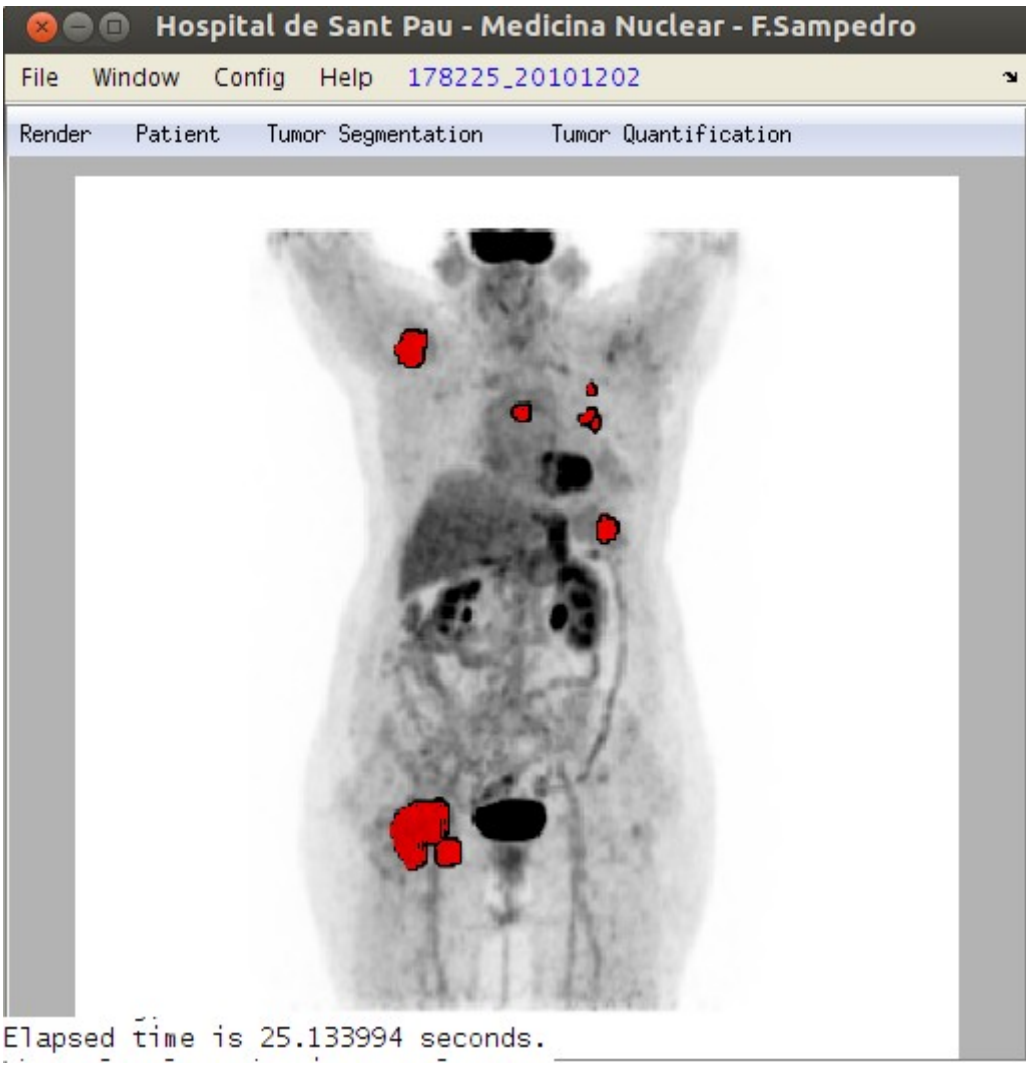


Fig.38 – Volum segmentat en inicialització amb Shear-Warp

En la segona imatge ***(veure Fig.39)*** es mostra la imatge final del volum segmentat, obtinguda a partir del renderitzat mitjançant l'algorisme de Ray Casting de volum sobre la GPU i en mode MIP. Aquest, ha estat implementat i incorporat en aquest projecte per tal de reduir el cost temporal de l'anterior algorisme a la vegada que intenta evitar pèrdues significatives en la qualitat de la imatge.

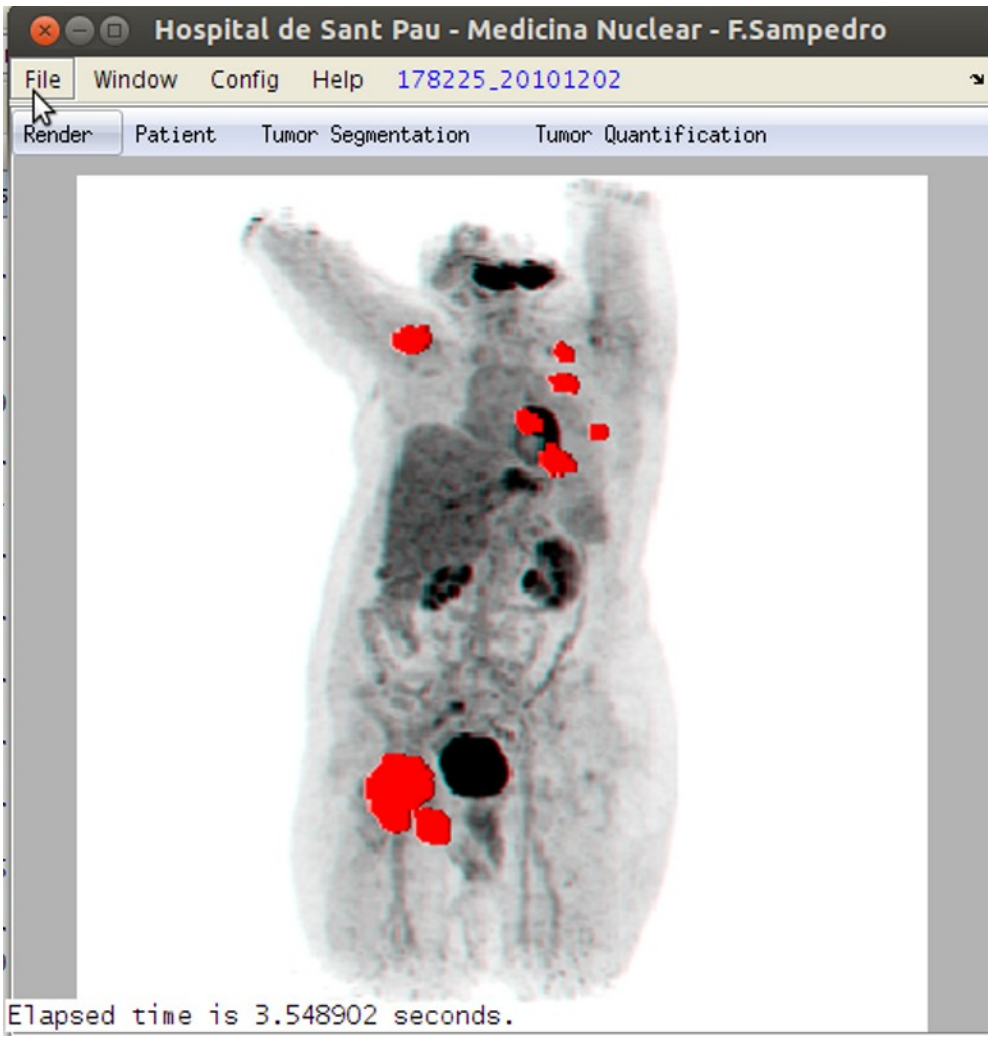


Fig.39 – Volum segmentat en inicialització amb Ray Casting GPU

Els resultats són molt satisfactoris ja que es pot apreciar com qualitativament no hi ha a quasi diferències entre un renderitzat i l'altre. La màscara de segmentació apareix assenyalant les mateixes zones patològiques en ambdues imatges i la visió del volum és pràcticament la mateixa. En la imatge obtinguda amb Ray Casting es poden apreciar més zones patològiques, que abans no eren visibles amb el ShearWarp. En conclusió, hi ha hagut gunays qualitatius significatius en aplicar el Ray Casting de volum sobre la GPU.

Per altra banda, pot observar-se com el cost temporal s'ha optimitzat considerablement respecte l'algorisme inicial del Shear-Warp. Si bé en aquest primer tarda uns 25 segons en mostrar el volum segmentat i incorporar-hi la màscara, en el Ray Casting de volum sobre la GPU el temps es veu reduït fins arribar a 3.5 segons. Això és una reducció temporal 7

vegades menor i suposa una gran millora que dóna una solució molt satisfactòria a la necessitat plantejada en la introducció del present projecte.

Les següents imatges ***(veure Fig.40 i 41)*** mostren el mateix volum segmentat que les dues anteriors amb els dos algorismes diferents però ara emprant la propietat de la interacció amb el volum un cop ja s'ha carregat i visualitzat aquest una primera vegada. Amb la interacció es pretén canviar el punt de vista del volum (tal i com es pot veure en ambdues figures ...) pel que només caldrà actualitzar o refrescar la imatge d'aquest sense dur a terme el procés de càrrega i inicialització de la primera vegada.

Fig.40 – Volums segmentats en interacció amb Shear Warp

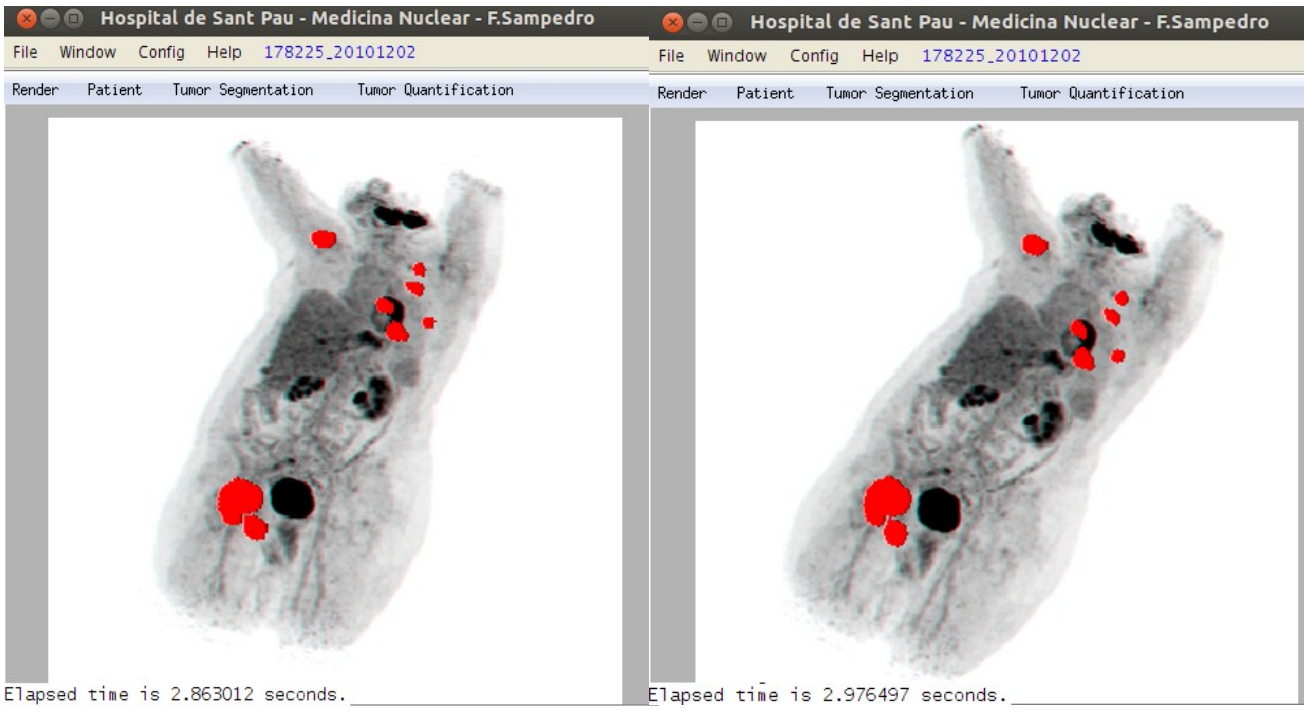


Fig.41 – Volums segmentats en interacció amb Ray Casting GPU

En aquesta ocasió, cal fixar-se com es redueix encara molt més que abans el cost temporal (fins a 2.9 segons aprox.) si s'usa la interacció amb l'algorisme del Ray Casting de volum sobre la GPU. Això es degut a que ja no caldrà dur a terme moltes de les funcions d'inicialització i càrrega que es realitzaven en la primera vegada de la crida de la funció, estalviant-nos molt més temps computacional. Per altra banda, en l'algorisme original de Shear-Warp, el temps per actualitzar la imatge després d'una interacció és d'uns 27-28 segons, pel que no tan sols es veu superat per l'algorisme del present projecte sinó que supera el cost temporal de la primera vegada de crida. En conclusió, la diferència és encara més remarcable i satisfactòria.

Els temps de refresc durant la interacció dependran de la duració d'aquesta (no tardarà el mateix moure el volum uns 20 graus que 360 graus).

Finalment, cal posar de manifest que tots aquests costos temporals anteriors podrien veure's disminuïts encara molt més si s'utilitzés un hardware (targeta gràfica, processador...) superior a l'emprat en aquestes simulacions, que ajudaria a augmentar les propietats de la

GPU del nostre algorisme dotant-lo de major velocitat a la simulada.

Simulacions amb el segon volum:

El volum utilitzat conté un món de vòxels de dimensions: 144x144x192

Es realitza el mateix procés que en les simulacions anteriors però ara amb un volum segmentat diferent i de dimensions més reduïdes .

Els resultats ***(veure Fig.42 i 43)*** són idèntics als donats en la simulació anterior tant per la interacció com per la primera visualització. El cost temporal en ambdós algorismes és un pèl inferior al de les simulacions anteriors, però això es deu a la reducció del món de vòxels del volum que genera menys temps de processament.

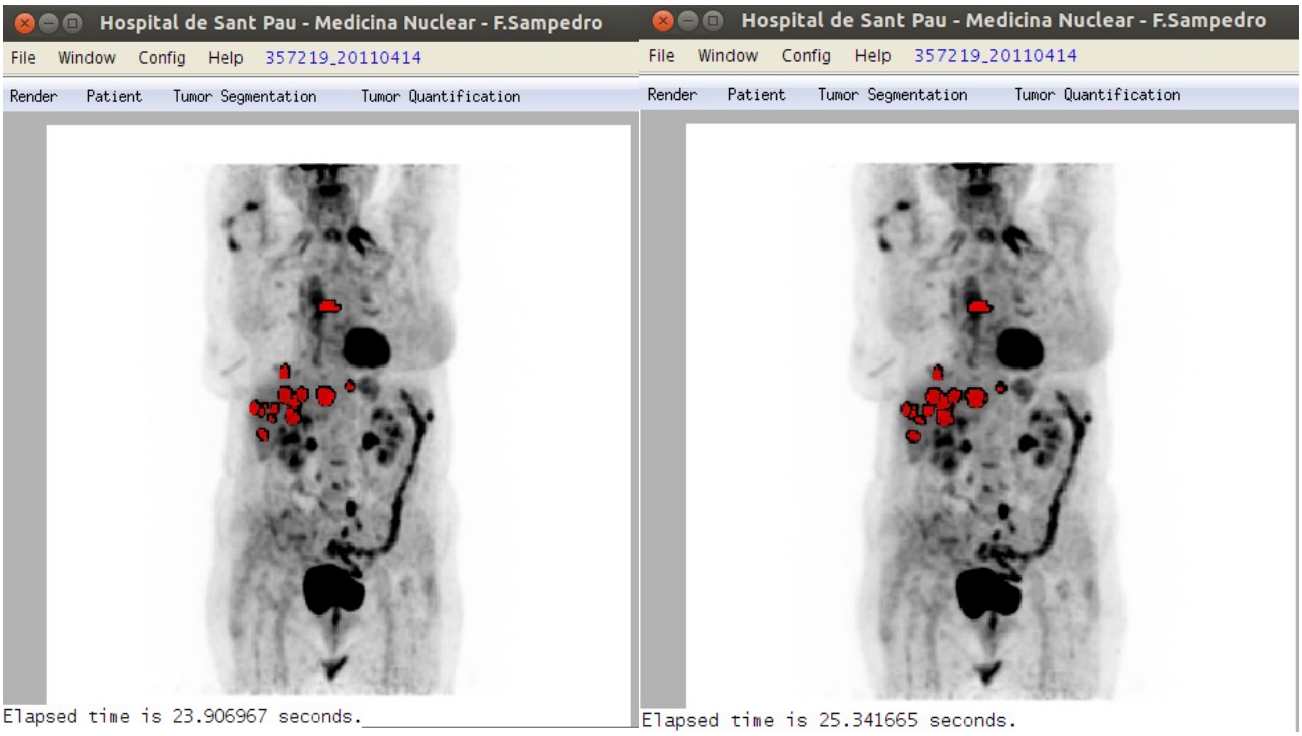


Fig.42 – Volum 2 segmentat en inicialització (dreta) i interacció (esquerra) en Shear-Warp

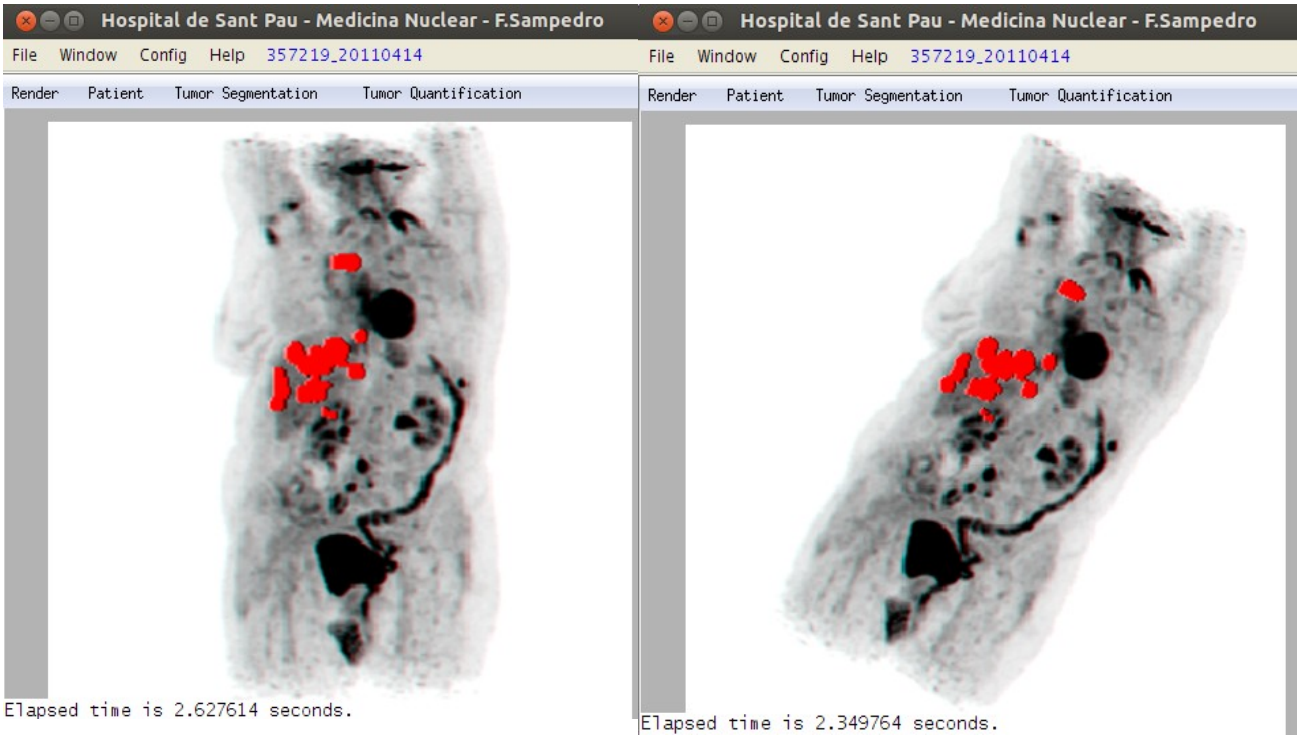


Fig.43 – Volum 2 segmentat en inicialització (dreta) i interacció (esquerra) en Ray Casting

6.2 Taula comparativa de temps:

En la taula que apareix a continuació ***(veure Fig.44)*** es mostra una comparació de diferents resultats temporals obtinguts en aplicar tant l'algorisme inicial Shear-Warp (que usa la CPU), com l'algorisme implementat en el present projecte del Ray Casting de volum (que usa la GPU).

Dimensions del volum i la màscara	*Algorisme de renderitzat emprat:*	*Basat en:*	*Funcionalitat:*	*Valor de mostreig:*	*Temps (s):*
144x144x213	*Shear-Warp*	*CPU*	*Inicialització*	----------	*26.411968*
144x144x213	*Shear-Warp*	*CPU*	*Interacció*	----------	*28.949338*
144x144x213	*Ray Casting*	*GPU*	*Inicialització*	*50*	*3.524712*
144x144x213	*Ray Casting*	*GPU*	*Interacció*	*50*	*2.623630*
144x144x213	*Ray Casting*	*GPU*	*Inicialització*	*30*	*4.516618*
144x144x213	*Ray Casting*	*GPU*	*Interacció*	*30*	*3.609096*
144x144x213	*Ray Casting*	*GPU*	*Inicialització*	*10*	*5.463438*
144x144x213	*Ray Casting*	*GPU*	*Interacció*	*10*	*4.431535*
144x144x192	*Shear-Warp*	*CPU*	*Inicialització*	----------	*23.933041*
144x144x192	*Shear-Warp*	*CPU*	*Interacció*	----------	*25.345362*
144x144x192	*Ray Casting*	*GPU*	*Inicialització*	*50*	*3.175711*
144x144x192	*Ray Casting*	*GPU*	*Interacció*	*50*	*2.345151*
144x144x192	*Ray Casting*	*GPU*	*Inicialització*	*30*	*4.280807*
144x144x192	*Ray Casting*	*GPU*	*Interacció*	*30*	*3.468453*
144x144x192	*Ray Casting*	*GPU*	*Inicialització*	*10*	*5.114120*
144x144x192	*Ray Casting*	*GPU*	*Interacció*	*10*	*4.312045*
144x144x237	*Shear-Warp*	*CPU*	*Inicialització*	----------	*27.565193*
144x144x237	*Shear-Warp*	*CPU*	*Interacció*	----------	*29.133891*
144x144x237	*Ray Casting*	*GPU*	*Inicialització*	*50*	*4.000597*
144x144x237	*Ray Casting*	*GPU*	*Interacció*	*50*	*3.006721*
144x144x237	*Ray Casting*	*GPU*	*Inicialització*	*30*	*4.827675*
144x144x237	*Ray Casting*	*GPU*	*Interacció*	*30*	*3.836391*
144x144x237	*Ray Casting*	*GPU*	*Inicialització*	*10*	*5.883740*
144x144x237	*Ray Casting*	*GPU*	*Interacció*	*10*	*4.934689*

Fig. 44 – Taula comparativa de costos temporals

En la taula mostrada, es diferencia la primera vegada de crida a la funció del renderitzat de les vegades en la què només s'ha dut a terme una interacció per canviar-ne els punts de vista. Tanmateix, es mostra la variació temporal respecte diferents mides de volums i amb valors de mostreig (sampling) i saturació diferents.

Analitzant la taula podem observar com:

- L'algorisme de Shear-Warp és temporalment molt més costós (unes 7 vegades més) que el Ray Casting de volum en tots els àmbits establerts. Entre altres factors, es deu a que l'algorisme de ShearWarp està implementat sobre la CPU i l'altre sobre la GPU, que posseeix unes característiques essencials per visualitzar qualsevol tipus de dades en temps real.
- Com més petit és el món de vòxels del volum, menors són els costos de temps en ambdós algorismes, tot i que la diferència és petita.
- Quan es duu a terme una interacció on només es vol refrescar la visualització del volum des d'un punt de vista diferent, en l'algorisme de Shear-Warp el cost temporal és elevat (més encara que la primera vegada de visualització). Per contra, en el Ray Casting de volum sobre la GPU el cost temporal és bastant inferior (un segon aproximadament) respecte la primera vegada de visualització, per l'estalvi en processos de càrrega i inicialització.
- El valor de mostreig de l'algorisme del Ray Casting és un factor determinant en la velocitat temporal d'aquest. Això es causat degut a que aquest és l'increment amb què el raig de l'algorisme fa el seu pas dins el bucle per poder travessar els vòxels. Per tant, un valor més petit, implica un bucle de processat major i en conseqüència, un temps superior que per a valors més grans.
- La màscara patològica es visualitza més nítida en l'algorisme de RayCasting que no en l'algorisme de Shear-warp, donant informació més precisa en aquestes zones.

7. Conclusió:

L'objectiu principal d'aquest projecte es centrava en l'anàlisi, disseny i implementació del mètode de Ray Casting de volum en mode de projecció de màxima intensitat (MIP) sobre la GPU que permetés la connexió amb MATLAB mitjançant una llibreria externa. Com a últim pas, calia integrar la funció MATLAB del renderitzat resultant a una aplicació software de MATLAB ja implementada que contenia la interfície adequada per carregar i interactuar amb volums mèdics RMI-PET segmentats. Aquesta aplicació té la seva funcionalitat final en l'Hospital Sant Pau de Barcelona i els seus usuaris principals són metges especialistes en oncologia i medicina nuclear.

La necessitat del projecte era la d'accelerar el procés de visualització (tant en la inicialització com en la interacció en temps real) dels volums segmentats per tal de reduir els elevats costos temporals que contenia inicialment l'algorisme dut a terme en l'aplicació referida. Aquesta millora ens l'ofereix l'algorisme de Ray Casting de volum sobre la GPU que aprofita el paral·lelisme d'aquesta per a treballar amb grans volums de dades a gran velocitat.

Com s'ha pogut observar al llarg del desenvolupament de la memòria, tant l'objectiu inicialment platejat com la necessitat derivada d'aquest s'han resolt de forma exitosa. S'ha aconseguit implementar l'algorisme de Ray casting de volum sobre la GPU i en mode MIP en una llibreria externa que es crida des d'una funció de renderitzat de MATLAB. A més a més, la integració amb l'aplicació inicial s'ha efectuat sense problemes. Pel que respecta a la resolució del problema de la necessitat, si s'observen els resultats obtinguts en les simulacions exposades del capítol 6, es por apreciar com el cost temporal es veu dràsticament reduït en utilitzar l'algorisme Ray Casting en GPU en comparació amb l'original de l'aplicació (Shear-Warp). A tot això, cal sumar també que no s'observa cap tipus de pèrdua qualitativa en la imatge final en utilitzar el Ray Casting. En definitiva, els resultats obtinguts són molt satisfactoris i solucionen a grans trets el problema plantejat en la necessitat del projecte.

L'ànalisi, disseny i implementació del present projecte ha estat un continu procés d'aprenentatge que ha anat creixent exponencialment. Els coneixements necessaris per

implementar el Ray Casting de volum sobre la GPU no es van donar en l'assignatura de Gràfics i Visualització i la connexió d'aquest algorisme amb MATLAB excedia els coneixements adquirits en qualsevol assignatura impartida. Per tant, ha calgut estendre i ampliar els coneixements en l'àmbit del processament de gràfics per computador. Això, sumat amb l'escassa documentació existent referent a com connectar un visualitzador extern de C++ que usa OpenGL i es programa sobre la GPU amb MATLAB, la incapacitat dels shaders de la GPU per efectuar debugging i la manca de precisió de MATLAB a l'hora d'indicar un error intern de la llibreria C++ usada que obligava sempre a apagar-lo, ha derivat en un llarg i exhaustiu procés d'investigació i anàlisi.

Finalment, es mostren a continuació les possibles millores i línies de continuació del present projecte:

- Traslladar adientment la llibreria implementada i tots els recursos OpenGL que aquesta necessita al sistema operatiu Windows. D'aquesta manera l'algorisme de renderitzat implementat en el present projecte seria funcional en Windows que és el sistema operatiu que utilitzarà el personal mèdic en l'aplicació final.

- Integrar adequadament les propietats de la càmera de l'aplicació MATLAB de partida amb la llibreria externa del renderitzat de Ray Casting de volum sobre la GPU tal i com s'indica en el *subcapítol 5.4* de la implementació.

- Millorar el mode de projecció de màxima intensitat (MIP) utilitzat en el Ray Casting de volum. Tal com s'ha descrit en l'apartat 2.1.5 dels antecedents, aquest mètode té el desavantatge de no determinar la profunditat del vòxel obtingut i això pot provocar algunes imprecisions en la imatge final. Per tal de solucionar aquest problema caldria combinar a aquest mètode les propietats que ofereix el mètode DVR per determinar la profunditat del vòxel. La combinació resultant donaria lloc a al mètode MIDA (Maximum Intensity Difference Accumulation).

- Implementar un classificador que indiqui el grau de gravetat de l'estat d'un pacient segons les zones patològiques obtingudes en la màscara de segmentació.

- Aprofitar la implementació del Ray Casting de volum en la GPU per dur a terme diferents tècniques de segmentació i classificació d'imatges.

8. Referències bibliogràfiques:

[1] **Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation** [Philippe Lacroute and Marc Levoy] *Proc. SIGGRAPH '94*, Orlando, *Florida, July, 1994, pp. 451-458.*

[2] **C++** - - http://www.cplusplus.com/

[3] **Clustering -** http://en.wikipedia.org/wiki/Cluster_%28computing%29

[4] **GLSL -** http://www.opengl.org/documentation/glsl

[5] **GPU Ray Casting: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading** [Engel, Klaus; Kraus, Martin; Ertl, Thomas], *2001, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*

[6] **Gràfics i visualització de dades, Tema2: Arquitectures gràfiques** [Anna Puig]

[7] **Gràfics i visualització de dades, Tema3: Elements geomètrics i transformacions geomètriques: matriu model-view** [Anna Puig]

[8] **Image segmentation -** http://en.wikipedia.org/wiki/Image_segmentation

[9] **Instant Volume Visualization using Maximum Intensity Difference Accumulation** [Stefan Bruckner and M. Eduard Gröller], *2009*

[10] **Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL** [Edward Angel, Dave Shreiner], *2011*

[11] **MATLAB –** http://www.mathworks.es/

[12] **OpenGL -** http://www.opengl.org/

[13] **Qt SDK -** http://qt-project.org/

[14] **Ray Casting** - http://en.wikipedia.org/wiki/Ray_casting

[15] **Ray Tracing** - http://en.wikipedia.org/wiki/Ray_tracing

[16] **Renderització 3D** - http://es.wikipedia.org/wiki/Renderizaci%C3%B3n

[17] **Three-dimensional display in nuclear medicine** [Wallis, J.W.; Miller, T.R.; Lerner, C.A.; Kleerup, E.C.] *1989, IEEE Trans Med Imaging 8 (4): 297–303*

[18] **Thresholding -** http://en.wikipedia.org/wiki/Thresholding_%28image_processing%29

[19] **Treball Fi de Grau: Ray Casting de volum CPU vs GPU** [Sergi García Trancoso, Anna Puig Puig]*, gener 2013.*

[20] **Volume Ray casting** - **Display of Surfaces from Volume Data** [Marc Levoy], *IEE CG&A, May 1988.*

[21] **Volume Rendering** - http://www.byclb.com/TR/Tutorials/volume_rendering/ch1_1.htm

[22] **Voxel** - http://en.wikipedia.org/wiki/Voxel

[23] **Writing MATLAB C/MEX Code** [Pascal Getreuer], *abril 2010*

[24] **Z-Buffering -** https://en.wikipedia.org/wiki/Z-buffering

Apèndix A: Manual tècnic

A.1 Instal·lació, requeriments mínims i passos a seguir

– Els requeriments **software** mínims necessaris per a poder utilitzar aquesta aplicació són:

- *Sistema Operatiu:* En principi l'aplicació MATLAB pot obrir-se en qualsevol distribució de Windows (XP/Vista/7) i en Linux (provada amb Ubuntu 12.04, compatible també amb versions anteriors d'Ubuntu i possiblement amb altres distribucions Linux).

** La funcionalitat de l'aplicació centrada en el present projecte (Ray casting de volum MIP en GPU), només és funcional empíricament en el sistema operatiu Linux, en concret, s'ha provat sobre* ***Ubuntu 12.04.*** *Per a altres distribucions Linux o altres sistemes operatius serà necessària la realització d'una sèrie de passos diferents als que es comentaran en el manual tècnic, tot i que no es descarta la viabilitat i compatibilitat amb aquests.*

- ***Software base per l'aplicació:*** Es necessària la instal·lació del software **MATLAB** per a poder obrir l'aplicació software dels volums mèdics RMI-PET. MATLAB és a la vegada entorn de programació de càlcul i llenguatge de programació.

** Pels motius explicats anteriorment, es requereix de MATLAB per a la plataforma Unix. Les versions utilitzades han estat la 7.14 R2012a i la 7.9 R2009b, i és molt probable que funcioni també en versions anteriors i posteriors a aquestes dues.*

- *Software (IDE) bàsic per la llibreria:* A part de l'entorn que ofereix MATLAB, es necessita també un software d'entorn de programació que faciliti la creació i generació de llibreries en llenguatge C++ i contingui els frameworks necessaris per poder treballar amb OpenGL. L'IDE de **Qt Creator** és l'ideal.

** Ha estat utilitzat el **Qt Creator 2.6.2** basat en **Qt 5.0** en plataforma Unix. Per a altres versions també seria compatible.*

- *Compilador:* És necessària la disponibilitat d'algun compilador pel llenguatge C++. El més comú (i utilizat en el projecte) és el compilador **G++**. Per a treballar amb arxius C++ en MATLAB, es requereix de la disponibilitat de **MEX-files** (Matlab Executable external interface fuctions) que el proveeixen d'una interfície per poder compilar i programar funcions en el llenguatge C/C++, sempre que es disposi d'un compilador instal·lat d'aquest llenguatge que ells mateixos busquen.

** El G++ ha de ser compatible en Linux (en concret, Ubuntu). Es recomana utilitzar la versió més actual.*

- *Llibreries:* Es requereixen totes les llibreries de C++ bàsiques per a compilar aplicacions en aquest llenguatge. Tanmateix, són necessàries les llibreries (i headers respectius) subministrades pel framework d'OpenGL, sent imprescindible el conjunt de llibreries oferides per **FreeGLUT3** i la llibreria de **GLEW**.

** Les llibreries han de ser compatibles amb el sistema operatiu Linux, per tant la seva extensió ha de ser de (.so*).*

- *Controladors de dispositiu:* És recomanable actualitzar els controladors de dispositiu de la targeta gràfica posseïda en la seva versió més recent per tal d'evitar problemes d'incompatibilitat amb el GLSL i la GPU.

– Els requeriments **hardware** mínims necessaris per utilitzar l'aplicació són**:**

- *Memòria RAM:* És compatible qualsevol memòria RAM que s'ajusti a les demandes del mercat actual i que no superi un llindar d'obsolescència.

** La memòria utilitzada en l'ordinador sobre el que s'ha dut a terme el projecte ha estat una de 2GB/Go DDR2 SDRAM.*

- *Processador:* És compatible qualsevol processador que s'ajusti a les demandes del mercat actual i que no superi un llindar d'obsolescència. També ho és tant per a processadors de 32 bits com de 64 bits, sempre que es tingui el software pertinent per a cada cas.

** En l'ordinador on s'ha provat el projecte hi havia un processador: Intel Dual-Core Processor T2370 de 1.73 Ghz. Un processador més bo acceleraria els resultats finals obtinguts mentre que un de més dolent els ralentitzaria.*

- *Disc Dur:* És estrictament necessari posseir un disc dur que contingui l'espai lliure suficient per a poder encabir tots els requeriments de software anteriors per tal de no tenir problemes d'espai en disc.

- *Targeta gràfica:* Es necessita disposar d'una targeta gràfica que sigui compatible, com a mínim, amb OpenGL 2.0 ja que la versió de shaders mínima requerida és la de GLSL 1.2. Targetes gràfiques compatibles amb versions infeiors d'OpenGL 2.0 tindrien una GPU incompatible per utilitzar l'aplicació, per contra, com més alta sigui la versió de shaders i OpenGL compatible, s'obtindran resultats temporals més alts.

** La targeta gràfica utilitzada per a totes les proves i simulacions del projecte ha estat una: Intel Mobile GM965/GL960 ,compatible amb OpenGL 2.1 i versió de shaders 1.2. Es podria utilitzar qualsevol marca de targeta gràfica (Nvidia, ATI, Intel) sempre que fos compatible amb OpenGL 2.0. Per saber si la nostra targeta gràfica compleix els requisits mínims , caldrà mirar les seves prestacions (per exemple, amb la comanda "glxinfo" en la terminal d'Ubuntu) i actualitzar els seus controladors.*

Per a dur a terme la instal·lació dels elements dels requeriments software anteriorment llistats cal seguir les següents indicacions:

- Per obtenir i instal·lar el sistema operatiu Ubuntu, anar a l'enllaç web:

http://www.ubuntu.com/download, triar la versió més actual que s'ajusti al hardware posseït i seguir les indicacions de la instal·lació.

- Per a obtenir i instal·lar MATLAB sobre el sistema operatiu Linux es pot anar a la pàgina oficial: http://www.mathworks.es/downloads/ si es posseeix d'una llicència.

- Per a obtenir i instal·lar Qt Creator i totes les seves llibreries i frameworks caldrà anar a: http://www.qt-project.org/downloads, escollir la versió més recent, i la plataforma SO d'interès. Posteriorment, caldrà seguir les instruccions del manual d'instal·lació propi de la web.

- Pel compilador G++ es recomana utilitzar el gestor de paquets Synaptic propi d'Ubuntu per a obtenir-lo, ja sigui via interfície gràfica o via comanda per consola ("sudo apt-get install package").

- Els MEX-files s'instal·len per defecte juntament amb la instal·lació de MATLAB. Per comprovar que funciona correctament i troba algun compilador per a C++ caldrà fer la comanda: "mex -setup" en la terminal pròpia de MATLAB i seguir les instruccions. Per a qualsevol altra informació, cal mirar la documentació del tutorial: http://www.mathworks.com/matlabcentral/fileexchange/27151-writing-matlab-cmex-code

- Les llibreries bàsiques necessàries tant de C++ com d'OpenGL, estan ja contingudes en la instal·lació de l'IDE Qt Creator. En cas de mancança d'alguna d'elles (freeglut3, GLEW o altres) es recomana utilitzar el gestor de paquets Synaptic per a obtenir-les.

- Per actualitzar els controladors de dispositiu de la targeta gràfica es pot anar a la pàgina web oficial de la nostra marca de targeta i cercar els controladors de dispositu més recents del nostre model de targeta gràfica.

  Per a Nvidia: http://www.nvidia.es/Download/

  Per a ATI: http://support.amd.com/us/gpudownload/

  Per a Intel: https://downloadcenter.intel.com

A.2 Manual del desenvolupador

Pel desenvolupament i testeig del projecte i l'aplicació, s'han utilitzat els requeriments software i hardware mencionats en l'apartat A.1 del present apèndix.

Suposant que el desenvolupador posseeix els requeriments hardware mínims necessaris i que ha pogut descarregar i instal·lar-se tots els requeriments software llistats sense problemes i de forma exitosa, els seus passos a seguir per armar correctament la part de l'aplicació que mostrarà de es poden ramificar en dues parts:

1. En primer lloc, haurà d'utilitzar l'eina esmentada d'entorn de programació C++, Qt Creator, i compilar i generar la llibreria que conté les funcions necessàries per utilitzar el renderitzat de volum en Ray Casting mode MIP sobre la GPU.

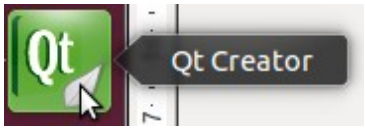
2. El següent pas (derivat del primer) serà connectar la llibreria generada referida i altres llibreries d'OpenGL a l'entorn de MATLAB per tal de poder establir la connexió MATLAB-OpenGL, compilar correctament tots els seus arxius MEX i linkar totes les llibreries necessàries.

** Per a la segona part, es necessitaran realitzar diverses comandes des de la terminal d'Ubuntu. Suposarem que el desenvolupador té nocions sobre comandes VI en Ubuntu.*

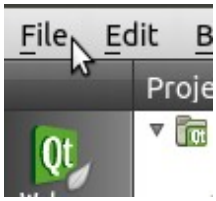
A.2.1 Passos amb entorn Qt Creator: Generació de la llibreria

Havent instal·lat correctament l'IDE de Qt Creator i totes les seves llibreries, compiladors i recursos necessaris esmentats sense problemes, es procedirà a realitzar els passos que generaran la llibreria externa de C++ que realitza l'algorisme de Ray Casting de volum sobre la GPU.

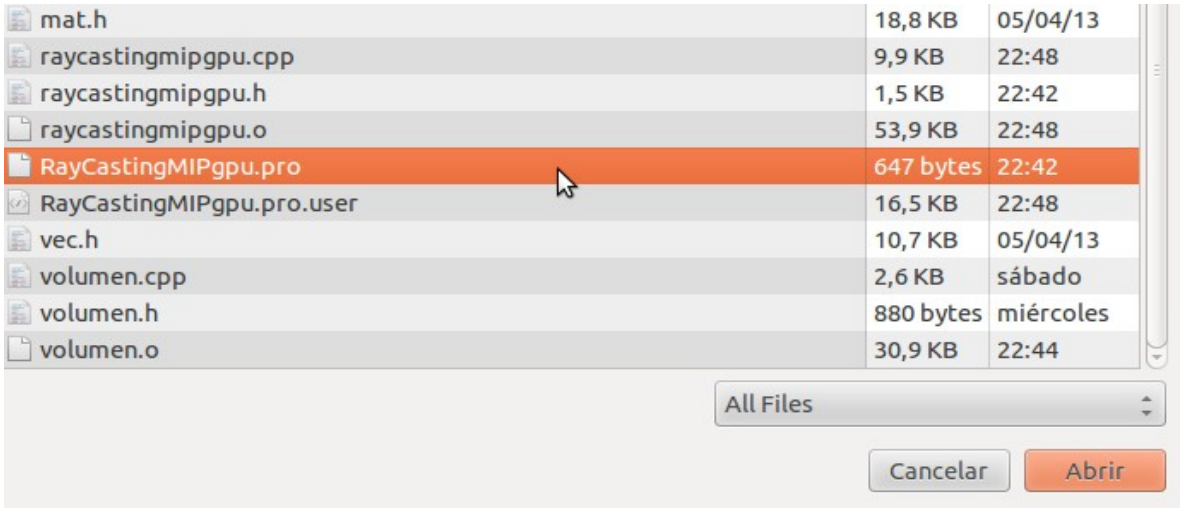
a) En primer lloc, obrim l'IDE de QT Creator amb dos clics a la seva icona (ja sigui al menú, ja sigui com a accés directe a l'escriptori).



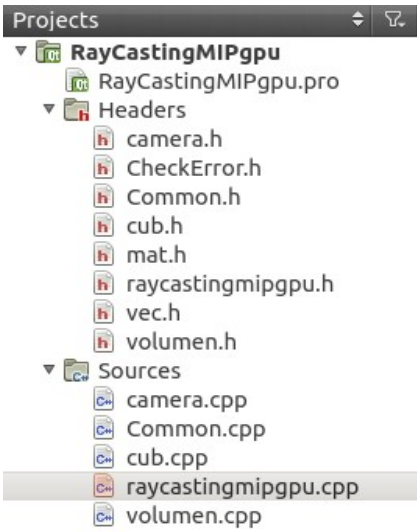
b) Ara s'ha d'obrir el projecte que conté la implementació de la llibreria, contingut en la carpeta *"Library"*. Per obrir el projecte, cal anar al menú superior, prémer "*File"* i després l'opció de "Open File or Project" del seu menú desplegable.



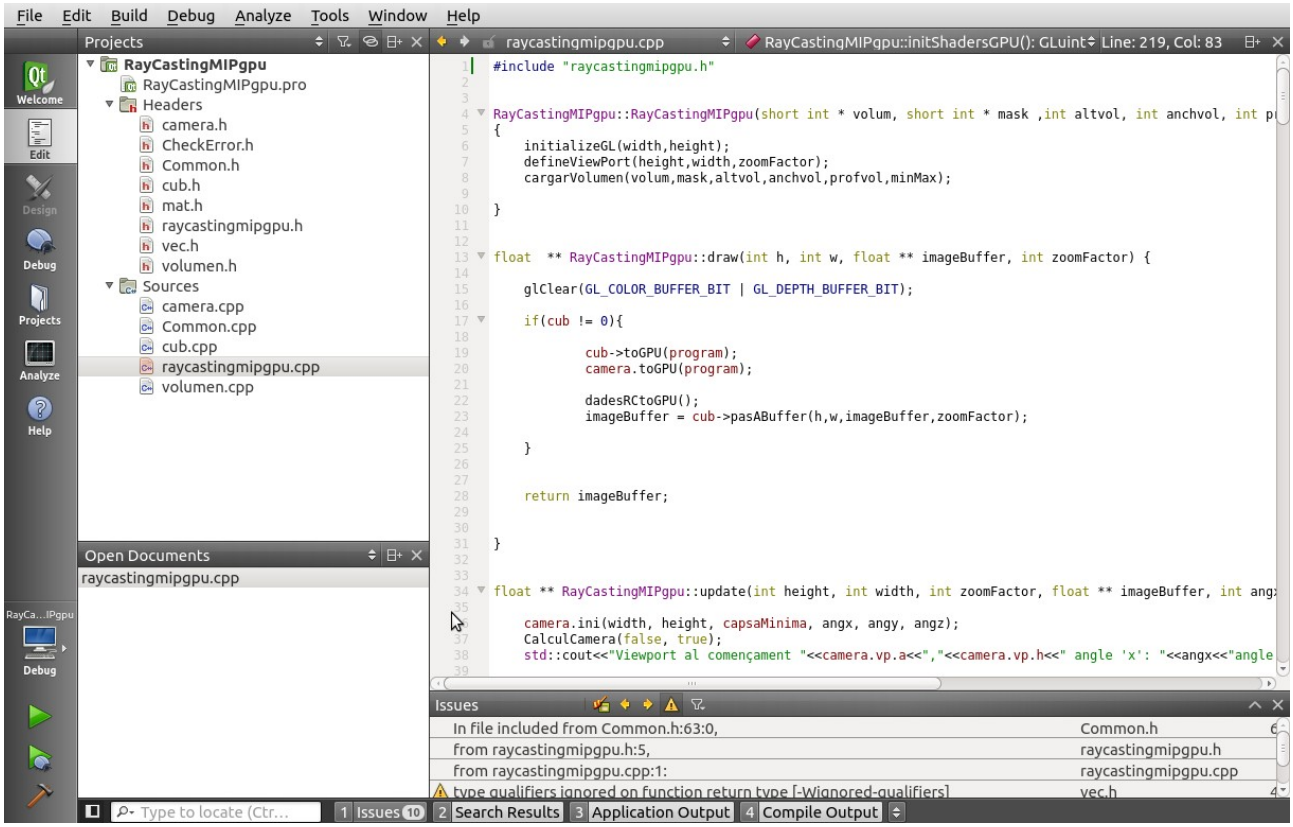
Posteriorment, anem cap on tinguem la carpeta "Library" , seleccionem l'arxiu central de càrrega "*RayCastingMIPgpu.pro*" (extensió '.pro') i l'obrim.



Se'ns carrega el projecte que conté la implementació de la llibreria. Despleguem les carpetes *"Headers"* i *"Sources"* i veiem, per una cantó, tots els els *"headers"* amb extensió ".h" utilitzats (que estableixen la connexió amb les fonts de codi i poden ésser importats) i per l'altre, totes les fonts de codi ".cpp".

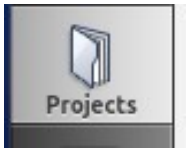


No és necessari, però si es vol veure el codi contingut en qualsevol dels arxius mostrats, amb un doble clic apareixerà a la finestra de l'esquerra.

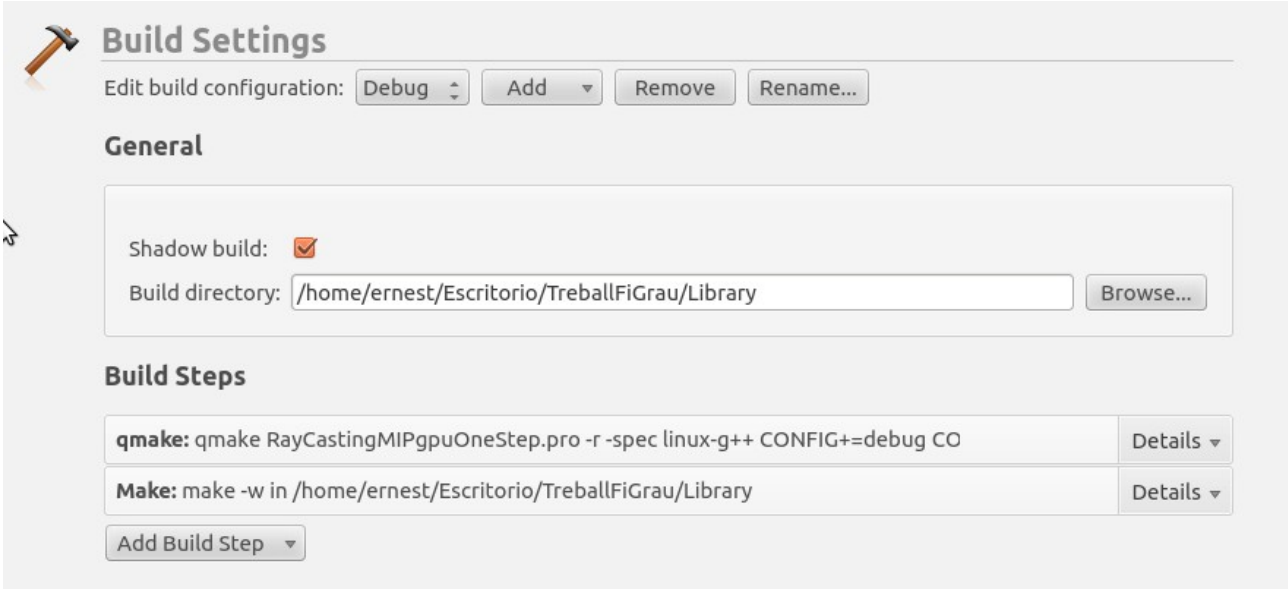


c) Un cop oberta la llibreria i abans de compilar-la, es recomanable configurar el path de la carpeta de compilació (ja que a vegades apareix desconfigurat). S'ha d'indicar que la carpeta on es durà a terme la compilació sigui la mateixa que conté la llibreria (Library), per evitar problemes posteriors.

Per fer la configuració, cal anar a la icona de *"Projects"* de la barra de menú vertical de la dreta. Un cop dins, ens hem de fixar en el contigut de *"Build directory"* que conté el path destí de la compilació i generació de la llibreria.



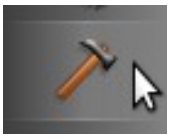
Si *"Build directory"* no concorda amb el path de la carpeta *"Library"*, es pot anar a *"Browse"* i seleccionar la carpeta desitjada per que el canvii.



Un cop canviat el path, ja tindrem la carpeta de compilació del projecte ben configurada. Si es vol tornar al lloc d'abans (no és obligatori), premem *"Edit"*.

d) Havent configurat correctament la carpeta de compilació, l'últim pas és compilar i generar la llibreria amb extensió ".so" per que pugui ser utilitzada des de l'aplicació de MATLAB.

Premem la icona inferior de la barra de menú vertical de la dreta de *"Build"* (Ctrl+B).



Fent això, es compila el projecte i es crea la llibreria. Si ha anat tot bé i no hi ha errors, comprovem que en la nostra carpeta *"Library"* hi ha arxius ".so" generats per veure si s'ha creat la llibreria.

A.2.2 Passos amb MATLAB: Connexió, compilació i linkat

Amb la llibreria generada correctament, toca realitzar els passos que enllaçaran MATLAB amb les llibreries d'OpenGL , amb la nostra llibreria, i realitzar la compilació i linkat final.

1.) Primer de tot, necessitem traslladar cap a MATLAB les llibreries OpenGL (i derivades d'aquestes) de: *GL, GLU, GLUT i GLEW*, i els seus headers associats (*gl.h, glu.h, glut.h i glew.h*). Això és necessari ja que aquestes llibreries s'utilitzen internament en els scripts de MATLAB i els MEX-files associats per connectar i obrir el context amb OpenGL i el widget iconificat on es pintarà el resultat final. Totes són compilades i linkades posteriorment en un script de MATLAB.

1.1) El primer subpas és accedir a la carpeta que conté el nostre programa MATLAB instal·lat. Normalment sol dir-se "MATLAB".

Tots els headers de MATLAB associats a llibreries externes i que es poden importar i compilar en MEX-FILES de C++ a partir d'includes, es troben al path *"MATLAB/extern/include/"* (suposant que estem dins la carpeta "MATLAB"). Per tant, tots els nostres headers hauran de posar-se dins aquest path.

Anàlogament, MATLAB també conté una carpeta on guarda totes les llibreries externes a ser linkades per MEX. Aquesta es troba en el path intern de: *"MATLAB/bin/glnx86"*, lloc on hauran de posar-se totes les nostres llibreries.

Finalment, per tenir els headers ben agrupats, crearem una carpeta dins de "*include*" que es digui "*GL*". Així, el path final on posar els nostres headers de les llibreries d'OpenGL serà: *"MATLAB/extern/include/GL*". Aquest pas és necessari ja que internament en el codi es realitzen els includes d'OpenGL a partir de *<GL/nom_header.h>*.

1.1.1) Per crear la carpeta GL, s'ha d'accedir a *"MATLAB/extern/include"* i dins

d'aquesta última carpeta, crear la nova. Si es demanen permisos de superusuari (altament probable) haurem d'accedir a la terminal d'Ubuntu, moure'ns cap al path *"MATLAB/extern/include"* i un cop allà, crear la carpeta en mode superusuari amb instruccions VI, per exemple: *sudo mkdir GL*

1.2) Realitzat i entès el subpas anterior, ara s'hauran de cercar els headers esmentats: gl.h, glu.h, glut.h i glew.h. Per cercar el tres primers, caldrà buscar per dins la carpeta on s'ha descarregat l'IDE Qt Creator. Pel "*glew.h*" haurem d'accedir al lloc on l'haguem descarregat.

Un cop trobats, només cal copiar-los cap a la carpeta destí. Segurament es necessitaran permisos i per tant, des de la terminal realitzarem la instrucció:

sudo cp <path origen><path destí >

On l'origen serà el path d'on hem trobat el header i el destí serà el de *"MATLAB/extern/include/GL/"*.

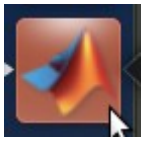
1.3) De la mateixa manera que en el subpas **1.2,** es cercaran les llibreries necessàries pel trasllat a MATLAB: *GL,GLU,GLUT i GLEW*. Les llibreries *GL,GLU* i *GLUT* es trobaran per dins d'alguna carpeta continguda en la que es troba l'IDE de Qt Creator. La *GLEW* es trobarà allà on s'hagi descarregat.

Un cop trobades les llibreries i els seus derivats (s'entén derivat com a un número després de l'extensió '.so', com per exemple: '.so.1.5'), es realitzarà la mateixa comanda que en el subpas anterior per fer el trasllat de cada una d'elles amb els seus derivats des del path d'origen cap al path destí de MATLAB, que en aquest cas seria: *"MATLAB/bin/glnx86"*.

2.) Havent fet el trasllat dels headers i les llibreries anteriors, el següent pas és el trasllat dels elements de la nostra llibreria, la que s'ha generat en l'apèndix **A.2.1**. D'aquesta manera, podrem linkar-la a MATLAB i cridar l'algorisme de Ray Casting de volum en GPU.

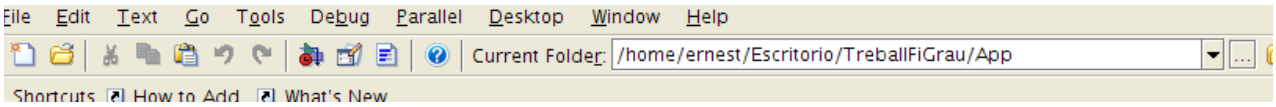
Anàlogament als subpassos **1.2** i **1.3**, copiem amb la comanda ja descrita tots el headers de la nostra llibreria cap al destí *"MATLAB/extern/include"* (alerta, que aquests no s'inclouran dins la carpeta GL de l'include!). Després, la llibreria ".so" generada i els seus derivats es copiaran de la mateixa manera cap a *"MATLAB/bin/glnx86"*.

Amb això, ja tindrem traslladades totes les llibreries (i headers) externes necessàries per compilar i linkar el nostre algorisme amb l'aplicació MATLAB.

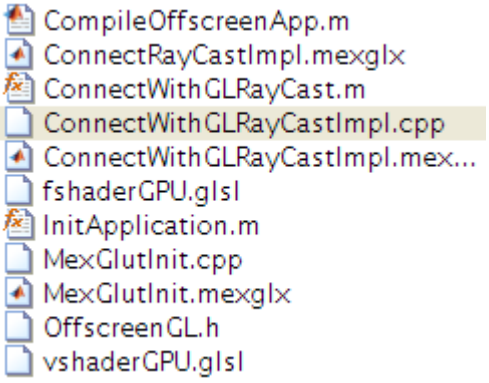
3.) L'últim pas,i el definitiu, consisteix en anar cap on tinguem el programa MATLAB i obrir-lo (ja sigui per comanda o per icona). 

Amb MATLAB obert, posem a *"Current Folder"* el path on tenim la carpeta "App" *(1)* que contindrà la MEX-Function principal que importa i crida la nostra llibreria *"RayCastingMIPgpu"* conjuntament amb altres funcions necessàries per a la connexió amb OpenGL, amb el fragment shader i amb el vertex shader (amb un doble clic sobre ells es podrà veure el seu contingut) *(2)*.

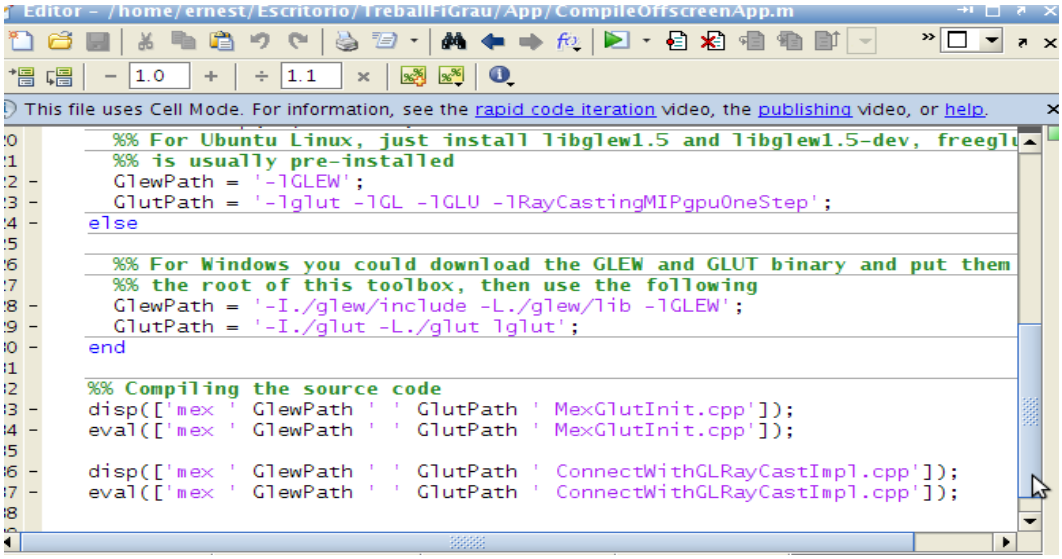
(1)



(2)



Per dur a terme la compilació i el linkat de tots els headers i llibreries traslladades necessàries per executar l'aplicació, caldrà executar l'arxiu "*CompileOffscreenApp.m*" escrivint el seu nom sense l'extensió ".m" a la terminal de comandes de MATLAB. Aquest script de MATLAB conté les comandes necessàries per linkar les llibreries: *GL, GLU, GLUT, GLEW* i la *RayCastingMIPgpu*, i per compilar els MEX-Files que les importen i utilitzen.

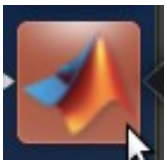


Un cop executat, si tot s'ha efectutat de forma correcta, no s'hauria de generar cap error. Les llibreries externes haurien quedat linkades i els MEX.files ".cpp" s'hauran compilat correctament generant un arxiu ".*mexglx*". Aquesta és la senyal què està tot preparat per executar l'aplicació sense problemes.

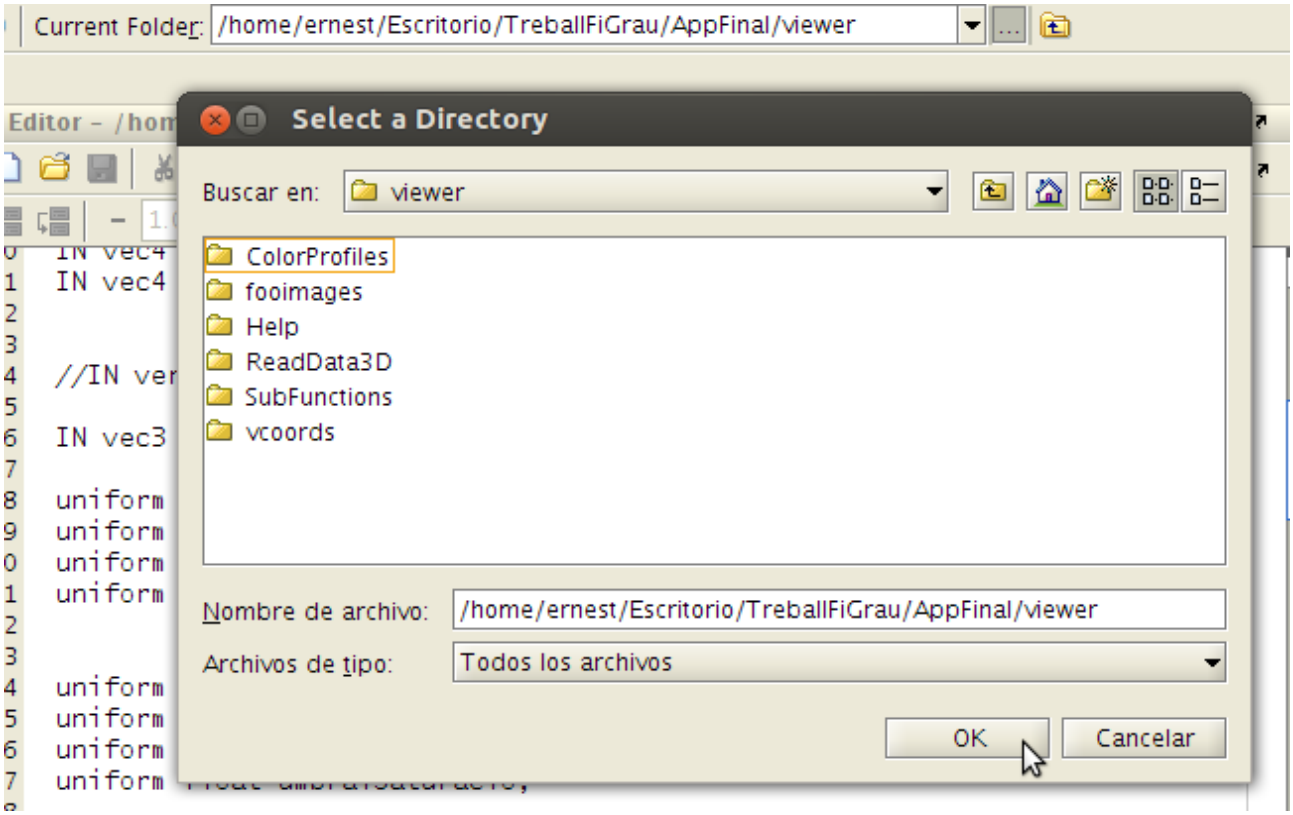
```
>> CompileOffscreenApp
mex -lGLEW -lglut -lGL -lGLU -lRayCastingMIPgpuOneStep MexGlutInit.cpp

Warning: You are using gcc version "4.6.3-1ubuntu5)".  The version
         currently supported with MEX is "4.4.6".
         For a list of currently supported compilers see:
         http://www.mathworks.com/support/compilers/current_release/

MexGlutInit.cpp: En la función 'void mexFunction(int, mxArray**, int, const mxArr
```

Apèndix B: Manual de l'usuari de l'aplicació

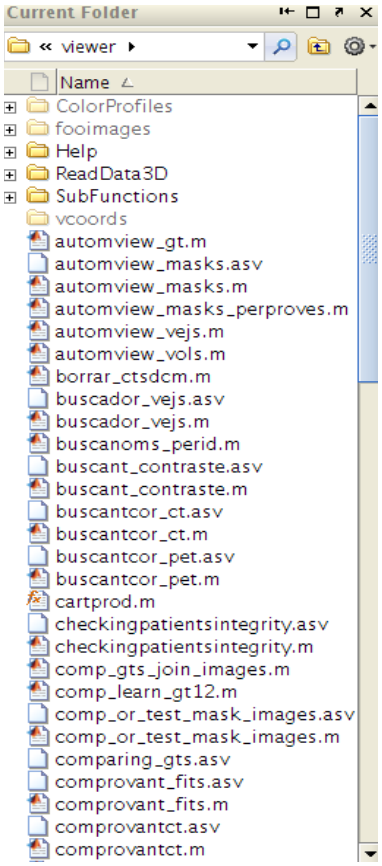
Per a poder executar i interactuar amb l'aplicació dels volums RMI-PET el primer pas és obrir el programa MATLAB amb dos clics sobre la seva icona o qualsevol altre mètode (terminal).



Un cop obert MATLAB, cal anar cap on tinguem la carpeta "AppFinal" que conté l'aplicació. Per dur a terme aquesta tasca, cal anar a "Current Folder" ,canviar el path on ens trobem i posar el path d'on tinguem la carpeta *"AppFinal/viewer"*.

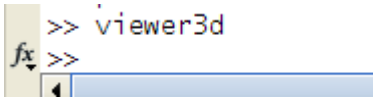


Havent carregat la carpeta *"viewer"*, el path de *"Current Folder"* quedarà canviat amb el que es desitjava. Es carregaran tots els arxius que conté, entre ells el d'iniciar l'aplicació.

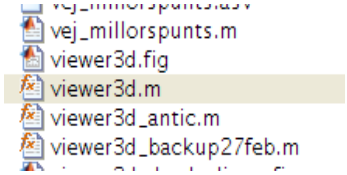


L'arxiu que inicia i executa l'aplicació és el denominat "*viewer3d.m*", per tant, per executar l'aplicació podem realitzar una de les següents opcions:

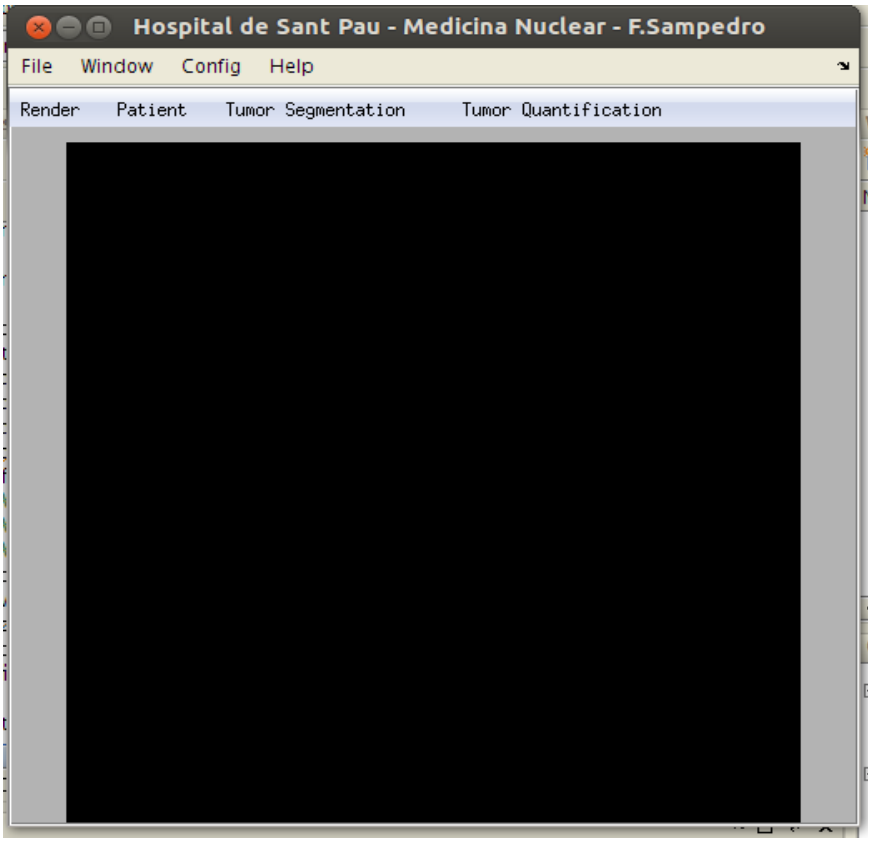
- Escriure en la terminal de comandes de MATLAB: viewer3d (sense l'extensió)



- Clicar dues vegades sobre l'arxiu "viewer3d.m" i prémer "F5"

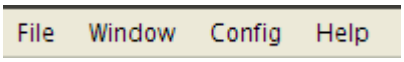


Si tot va bé, s'executarà l'aplicació i apareixerà la seva interfície gràfica.



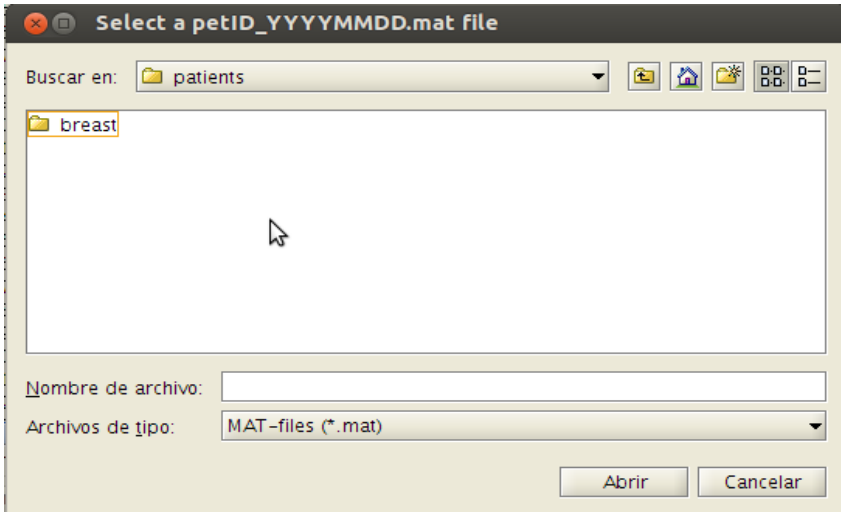
Si es vol, pot maximitzar-se prement: 

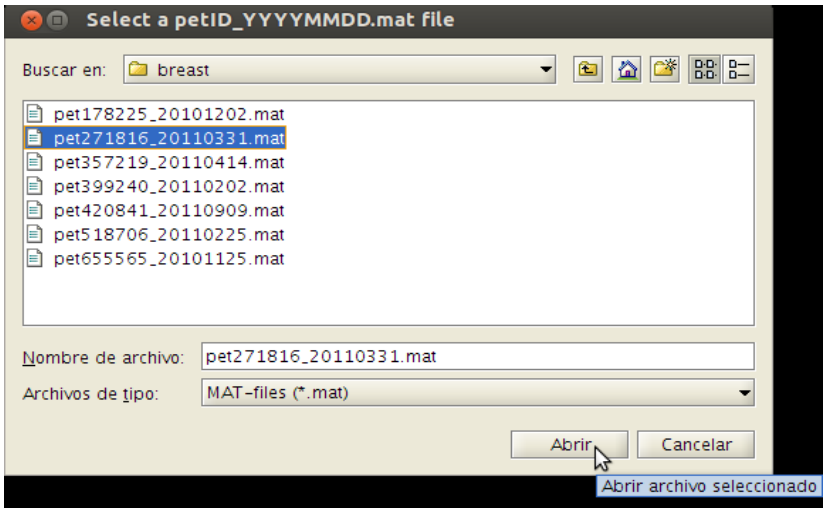
Amb l'aplicació oberta, el primer pas de tots serà carregar un volum mèdic RMI-PET dels existents dins d'aquesta. Per aquesta tasca cal:

a) Prémer l'opció “File” de la barra de menú superior. 

b) En el menú desplegable d'aquest, prémer l'opció “Open Medical 3D File (.mat)”.

c) Entrar dins la carpeta “breast” amb dos clics i seleccionar qualsevol dels volums que hi ha en el seu interior (arxius amb extensió '.mat').





d) Obrir el volum

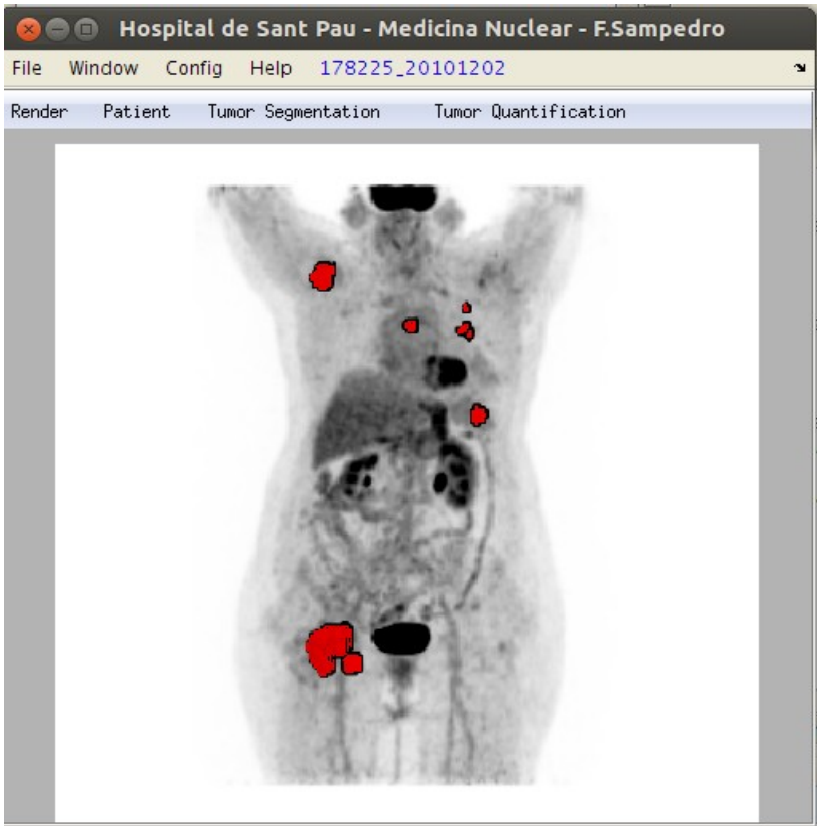
Seguidament, es visualitzarà el volum mèdic carregat anteriorment.

Aquest, apareix per defecte sense la seva màscara de segmentació. Per aplicar la màscara i veure el llarg temps que tarda, caldrà:

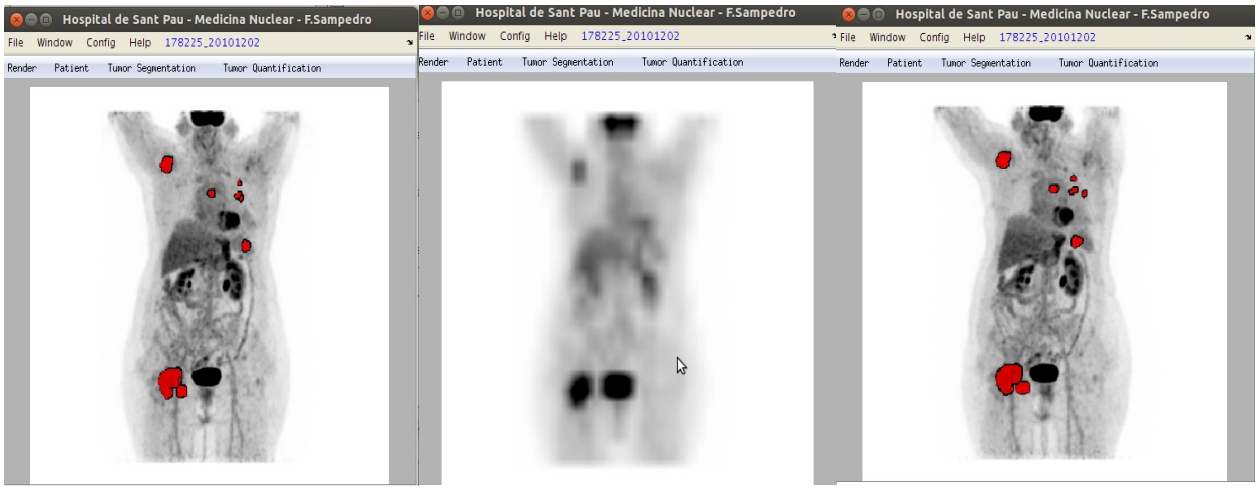
a) Anar a l'opció "*Tumor Segmentation*" de la segona barra de menú.

b) En el menú desplegable derivat d'aquesta opció, prémer "*Show/Hide Mask*".

c) Esperar uns segons fins veure com el volum de la imatge queda segmentat en zones hipotèticament patològiques de color vermell.



Ara ja es té el volum mèdic RMI-PET carregat i segmentat. Aquest volum està mostrat a partir de l'algorisme "Shear Warp" (ja implementat prèviament a la realització del projecte) en mode MIP, que dóna uns resultats molt lents. Si es vol interactuar amb el volum i moure'l (rotar-lo), només cal fer un clic sobre algun put de la imatge i moure el ratolí segons la direcció cap on el vulguem rotar (esquerra o dreta). Es pot observar que els temps d'actualització i recàrrega són encara més lents.

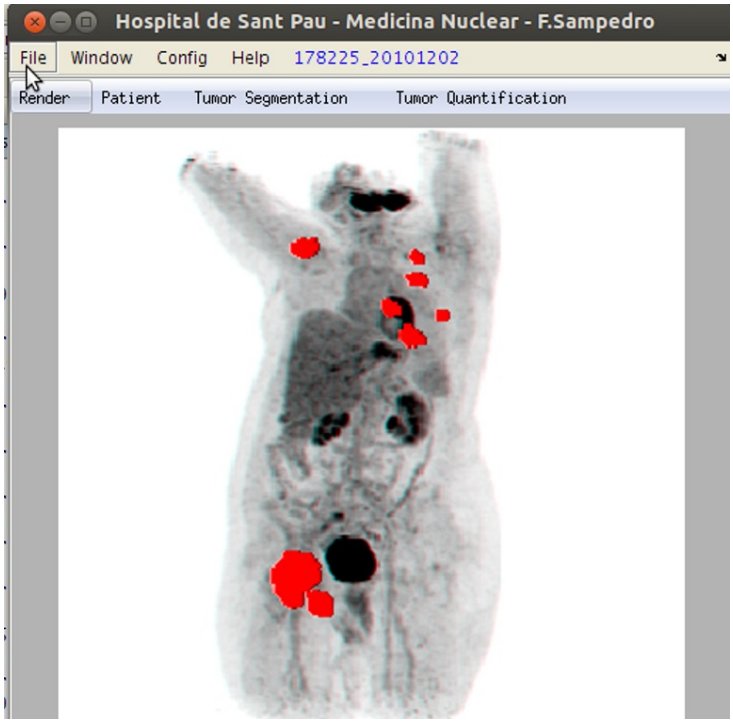


Per tal de poder veure el volum mèdic RMI-PET segmentat utilitzant la funcionalitat del Ray Casting de volum sobre la GPU i en mode MIP, que és la base d'aquest projecte, caldrà seguir els següents passos:

a) Prémer l'opció *"Render"* de la segona barra de menú.

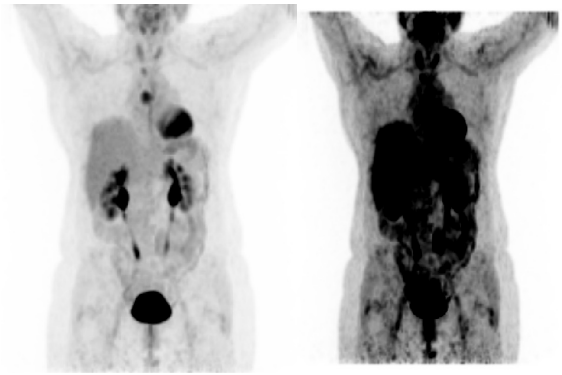
b) En el menú desplegable derivat del "Render" seleccionar l'opció *"GPU MIP"*.

Es veurà el mateix volum utilitzant el nou algorisme extern. Ara, es pot apreciar com el temps de càrrega és molt més ràpid que abans. Si es vol interactuar amb el volum, es realitza el mateix procés d'abans i s'observarà també que els temps d'actualització són molt millors.

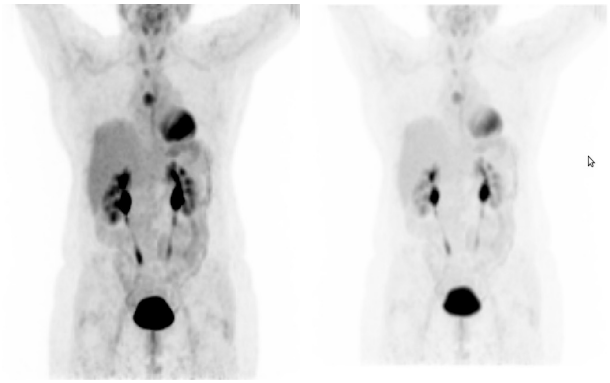


Independentment de l'algorisme de render utilitzat per mostrar el volum mèdic, existeix també una funcionalitat addicional que ofereix la possibilitat de modificar el contrast de la imatge del volum i visualitzar-ho en la interfície. Si es disposa d'un ratolí amb rodeta "scroll", només caldrà moure la rodeta i el contrast de la imatge s'anirà modificant:

a) Moure l'scroll cap avall, fa augmentar el contrast gradualment
(a més moviments d'scroll, més anirà augmentant)



b) Moure l'scroll cap amunt, disminueix el contrast gradualment



Finalment, en cas de voler sortir de l'aplicació només caldrà tancar la interfície prement: 

** L'aplicació utilitzada conté moltíssimes altres opcions i funcionalitats a realitzar, però aquest manual està centrat només en l'àmbit del present projecte, i per tant, en la càrrega i renderitzat d'un volum segmentat mitjançant el Ray Casting de volum en mode MIP sobre la GPU i l'actualització o moviment d'aquest a partir de la interacció i la comparació d'aquest amb l'algorisme anterior.*