Renderització d'imatges mèdiques en iOS mitjançant OpenGL i ITK

Memòria del Projecte Final de Carrera

Albert Plana Padró

18 de juny del 2013

Director: Hugo Hernández Pibernat
Ponent: Ricard Gavaldà Mestre
Departament del ponent: LSI
President: Carlos Antonio Andújar Gran
Vocal: Lídia Montero Mercadé
Titulació: Enginyeria Superior Informàtica
Centre: Facultat d'Informàtica de Barcelona (FiB)
Universitat: Universitat Politècnica de Catalunya (UPC) – Barcelona Tech

Agraïments

Normalment aquesta secció és més una formalitat que una altra cosa. En el meu cas, no són paraules buides, són l'expressió de la gratitud que sento per les persones que m'han ajudat a fer-lo possible. Aquest projecte ha dut molts mal de caps, i no hagués estat possible que acabés com ha acabat si no fos per elles.

Per començar, voldria agrair a les tantíssimes persones que m'han donat suport quan aquest projecte no avançava i jo em desesperava, que m'han recolzat quan jo no em veia capaç de tirar-lo endavant. En especial voldria agrair-ho als meus pares, amb el seu suport incondicional, i als meus amics. D'aquests últims caldria mencionar a masses noms, per ser just, per tant prefereixo no fer-ho. Tots han compartit les meves preocupacions i m'han animat quan realment ho necessitava.

També agrair al director del projecte, Hugo Hernández, tot el que ha fet per aquest projecte. Va haver un lapse de temps en que es va perdre la comunicació, però quan es va recuperar ell va fer tot el que era possible per poder presentar aquest projecte.

També agrair a en Joan Martín, una persona que sense tenir cap mena d'obligació, ha dedicat molt esforç en ajudar-me a tirar endavant el projecte.

Finalment agrair als membres del tribunal la seva paciència i col·laboració amb el veritable caos que ha estat coordinar la defensa d'aquest projecte, sobretot a la professora Lídia Montero, a la que vaig atabalar canviant la data múltiples vegades i no va presentar mai cap objecció.

Gràcies a tots.

Índex

1. Introducció

El món de la medicina és molt complex. No és fàcil esbrinar què pot patir un determinat pacient a partir d'uns quants símptomes, ja que a vegades les possibilitats no són poques.

Si més no, la tecnologia ha format part dels grans avenços en medicina dels darrers anys. Qualsevol eina que faciliti la composició d'un diagnòstic és ben rebuda.

Per exemple, tirant molt enrere, al 1985 Wilhelm Röntgen Röntgen, un físic alemany, va descobrir els Rajos X. Això va suposar una revolució en el món de la medicina, ja que obria un nou món immens de possibilitats.

Pel 1921, es va utilitzar per primer cop un microscopi en una operació. Actualment, això ha evolucionat cap a les anomenades endoscòpies. Aquestes es realitzen amb una lampareta col·locada a l'extrem d'un cable extremadament fi de fibra òptica. Gràcies a la invenció d'aquesta tècnica, s'han pogut practicar cirurgies amb un impacte molt menys agressiu al pacient.

Al 1942, s'utilitza per primer cop un ronyó artificial per realitzar diàlisis. No cal dir que també va suposar un gran avenç.

Per l'any 1952, s'implanta el primer marcapassos: un dispositiu electrònic que fa batre el cor a partir d'impulsos elèctrics. La bateria d'aquests dispositius dura més de 10 anys.

Queda més que constatat, doncs, que la tecnologia té moltíssim que oferir a la medicina. Un petit pas tecnològic pot significar un gran nombre de vides salvades, o sense ser tant dramàtics, millorades en qualitat de vida.

De fet, la inclusió de la informàtica en la medicina dóna un fort impuls a aquesta. La informàtica s'ocupa del recursos, dispositius i mètodes necessaris per a l'adquisició, emmagatzematge, recuperació i utilització de la informació de tipus biomèdica. Un cop adquirida, mantinguda i processada la informació mèdica, els responsables adients la tenen molt més a mà i en molt millors condicions que antany, i això els permet treballar molt més còmodament amb ella.

Hem vist exemples del passat, però un exemple del que es podria aconseguir en un futur, algun dia, és aplicar els coneixements en nanotecnologia per a la construcció i programació de nanobots, que busquessin i destruïssin les cèl·lules responsables de la generació de càncer. Fins i tot es podrien emprar per la reestructuració o reparació de teixits, músculs o ossos. Les fractures podrien ser cosa del passat: els nanobots podrien ser programats per detectar fissures en els ossos i arreglar-les de dues formes: accelerant la recuperació de l'os d'alguna manera o fundint-se amb l'os trencat.

I no només això, els nanobots podrien injectar antibiòtics o antisèptics, en certes zones del cos, per reduir els efectes de refredats, inflamacions, etc.

Així doncs, aquest projecte intenta contribuir en aquest ventall d'eines tecnològiques que intenten facilitar la feina d'un metge a l'hora d'estudiar un pacient. El fet de voler mostrar el resultat d'un escàner mèdic en un dispositiu que es pot dur a sobre en qualsevol moment, com és un iPhone o un iPad, pot suposar un avantatge clar davant l'opció de només disposar del resultat d'un escàner en un ordinador de sobretaula.

Ara bé, si bé és cert que la motivació principal d'aquest projecte és l'aplicació en sí, que sigui capaç de mostrar les imatges en tres dimensions[1], hi ha altres motivacions derivades, com per exemple aprendre els avantatges i inconvenients de la programació per dispositius mòbils (telèfons, tablets o fins i tot un reproductor MP5, com pot ser l'iPod Touch).

Una darrera motivació és pensar que és un projecte on l'èxit pot ser molt suculent: pot haver fins i tot la possibilitat de publicar l'aplicació desenvolupada en la botiga d'aplicacions d'Apple, l'App Store. I no només això, provocarà un gran sentiment de satisfacció de pensar, que en algun moment, potser algú utilitzarà l'aplicació desenvolupada i la trobarà útil. Només per això, ja val la pena intentar-ho.

[1] Més detalls en la següent secció.

1.2. Definició dels objectius

L'objectiu del projecte és programar una aplicació que corri sobre dispositius iOS que permeti mostrar imatges mèdiques en tres dimensions.

Per a tal aplicació, es vol utilitzar la llibreria de tractament d'imatges ITK, i per a la visualització en sí, es vol fer servir OpenGL ES, que és el subconjunt d'instruccions d'OpenGL disponible per dispositius mòbils.

A part de simplement mostrar la imatge, l'aplicació ha de permetre a l'usurari interactuar amb la imatge mostrada. Concretament, les opcions que ha de permetre l'aplicació són les que permetin una bona visualització d'aquesta:

- Zoom
- Pan[2]
- Rotació de la imatge en els 3 eixos x, y i z.

Aquestes navegacions s'han de poder fer principalment amb gestos tàctics sobre el dispositiu. És també un objectiu de valor afegit fer que els moviments es puguin fer utilitzant el giroscopi del dispositiu (amb inclinacions del dispositiu).

Addicionalment, i aprofitant el potencial de la llibreria ITK que es vol utilitzar, també es marca com a objectiu utilitzar la llibreria per aplicar filtres a la imatge a mostrar.

El que es pretén amb aquest projecte és facilitar considerablement la visualització de les imatges mèdiques resultants de, per exemple, ressonàncies magnètiques.

El motiu que justifica el voler fer que l'aplicació corri a dispositius mòbils és permetre el seu ús en qualsevol lloc.

Si bé això pot sonar obvi, cal destacar que al tractar-se d'una aplicació destinada a la medicina, "qualsevol lloc" podria incloure fins i tot l'interior d'un quiròfan, on sens dubte tenir una visió en 3D de la zona a on s'ha d'incidir pot ajudar. Que es puguin entrar o no dispositius com iPads en un quiròfan és un altre tema, recordem que diversos estudis[3] demostren que pot haver-hi 18 vegades més bactèries en la pantalla tàctil d'un dispositiu que en la cadena d'un lavabo d'homes públic. Probablement hi hagi algun mètode per assegurar l'esterilització del dispositiu per l'ús en quiròfan, però de no ser-hi, l'aplicació seguirà permetent visualitzar les imatges mèdiques a qualsevol lloc i moment, i encara que en un quiròfan no es pugui, ja serà útil.

Cal destacar que existeix una diferència entre les aplicacions dissenyades per iPod Touch o iPhone i aquelles que ho han estat per iPad, degut al notable canvi de mida de les pantalles. En el present projecte, es procurarà que l'aplicació es vegi correctament en ambdós dispositius, però de no poder ser, es prioritzarà el tipus iPod/iPhone, ja que és el tipus de dispositiu del que es disposa actualment.

---

[2] Desplaçament de la imatge relatiu als eixos de coordenades de dispositiu (de la pantalla de l'iPod, per exemple).

[3] Veure referència extra [1]

1.3. Metodologia

La metodologia a seguir durant el projecte es pot dividir en les següents fases:

En primer lloc, s'estudiarà una mica l'entorn de desenvolupament, les eines amb les que es treballarà, Xcode sobretot.

Posteriorment, s'estudiarà Objective-C amb l'objectiu d'aprendre les bases necessàries per desenvolupar l'aplicació.

Abans de començar la part més complexa del projecte, s'estudiarà com aplicar rotacions, zoom i les altres transformacions d'escena necessàries per visualitzar l'escena tal com s'havia descrit en els objectius.

A continuació, s'inclourà la llibreria ITK a l'aplicació construïda fins el moment.
En aquest moment, es podrà començar a intentar llegir i representar la imatge en 3D utilitzant OpenGL i la llibreria.

Per últim, quan tot això estigui en funcionament, es modificarà l'aplicació per permetre l'ús d'alguns filtres ITK que permetin visualitzar la imatge de diferents maneres, les que es creguin més convenients.

1.4. Planificació inicial

1.4.1 Planificació temporal

A priori és molt difícil planificar un projecte. En aquest cas, on es desconeix el 100% de l'entorn, encara resulta més incert si tot allò que es planificarà es podrà dur a terme dins els terminis proposats, o si hi haurà alguna dificultat que farà que alguna de les parts s'endarrereixi.

Tot i així a continuació s'adjunta un diagrama de Gantt amb el que s'espera que sigui el temps invertit en el projecte:

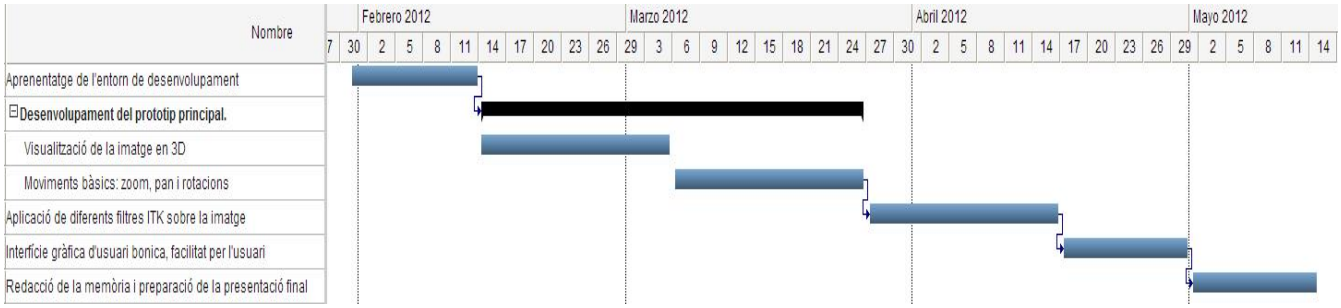


Figura 1.4.1.1: Diagrama de Gantt de l'estimació de la duració del projecte

Com es pot observar, es destinarien dues setmanes a l'aprenentatge de l'entorn de treball, així com una primera posada en marxa potser posant a prova tutorials de com programar per iOS.

Posteriorment, vindrien 6 setmanes per programar el que vindria a ser l'aplicació principal. 3 d'aquestes setmanes estarien destinades a la part de visualització de la imatge i 3 en l'aplicació dels moviments zoom, pan i rotacions. Aquest últim interval s'ha exagerat, ja que sigui probable que amb 3 setmanes no hi hagi prou per la part de la visualització.

A continuació, vindrien 3 setmanes en les quals s'aplicarien filtres de la llibreria ITK sobre la imatge.

Més endavant, vindrien 2 setmanes en les que es "posaria maca" l'aplicació, es faria una interfície gràfica d'usuari agradable i fàcil de navegar. Possiblement s'aprofitaria per optimitzar l'aplicació, si s'escaigués.

I ja per acabar, 2 setmanes amb les que es preveu redactar la memòria del projecte així com la preparació de la presentació final.

Així doncs, s'estima un projecte de 15 setmanes de duració: més o menys el temps mig que sol durar un projecte final de carrera.

1.4.2 Eines previstes

Deguda la naturalesa del projecte, les eines a utilitzar estan limitades a aquelles que Apple habilita per desenvolupar pels seus dispositius. En el cas concret d'aquest projecte es preveu fer servir:

- Un Macbook Pro (portàtil) de 13 polzades. Qualsevol altre màquina (ordinador) de la marca Apple també hagués servit (iMac, Mac mini, etc.) però actualment es disposa d'aquest. El sistema operatiu que duu és el Mac OSX Lion versió 10.7.3.
- Entorn de Xcode: IDE[4] que conté un seguit d'eines ideals per facilitar el desenvolupament d'aplicacions per Mac OSX o iOS.
- Simuladors: Juntament amb l'Xcode, per facilitar el desenvolupament d'aplicacions per a dispositius mòbils, es disposa d'uns simuladors d'iPhone i iPad que es preveuen fer servir per a fases inicials i abans de fer servir el dispositiu real. Concretament, es farà servir el simulador d'iPhone i iPad amb firmware 5.0.
- iPod Touch 4G: amb versió de software 5.0, del que es disposa actualment. Serà on es faran les proves en entorn real.

[4] IDE: *Integrated Development Environment* (Entorn de Desenvolupament Integrat): programa compost per un seguit d'eines que faciliten la programació d'una aplicació. Consta d'un editor de text pel codi, un compilador, un depurador (per debugar) i un constructor d'interfície gràfica.

2. Anàlisi d'antecedents i factibilitat

2.1 Antecedents

En el moment de començar el present projecte, es desconeixia l'existència d'aplicacions semblants per dispositius mòbils. De fet, resulta lògic pensar que com a mínim versió d'escriptori (PC) n'hi havia d'haver, però es desconeixia.

Fent un anàlisi més acurat, es van descobrir les següents aplicacions:

2.1.1 Voreen

Pàgina web: http://www.voreen.org/

És una aplicació de software lliure que utilitza una tècnica anomenada *volume rendering*[5] per representar dades volumètriques.

Nascuda al 2005 de la mà de la universitat de Münster és actualment mantinguda per una cooperació entre aquesta universitat i la universitat de Linköping.

És multiplataforma: apte per Windows i Linux.

Algunes de les possibilitats que ofereix aquesta eina:

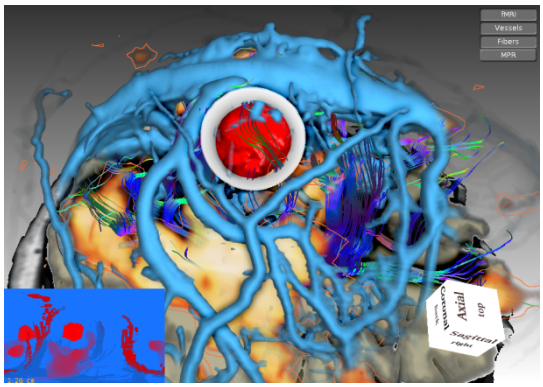


Figura 2.1.1.1: Eina per a planificació pre-neurocirurgia

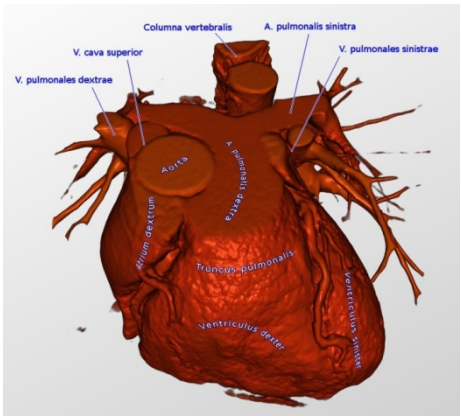


Figura 2.1.1.2: Etiquetes 3D

[5] Tècnica explicada a l'apartat 3.2.9.4.

2.1.2 STARVIEWER

Pàgina web: http://iiia.udg.edu/en/research/1-starviewer.html

Aplicació produïda per l'Institut d'Informàtica i Aplicacions de la Universitat de Girona.

És una plataforma visualitzadora d'imatges per tal de facilitar el diagnòstic mèdic.

És multiplataforma, apte per Windows i per Linux.

És una eina fàcil de fer servir. Permet personalitzar la interfície d'usuari, dissenyada en un principi d'acord amb les especificacions proporcionades per radiòlegs.

Té un seguit de funcions de tractament d'imatges que estan enllaçades a combinacions de teclat, de manera que siguin ràpides d'usar.

Té un disseny modular, amb el que és fàcilment ampliable.

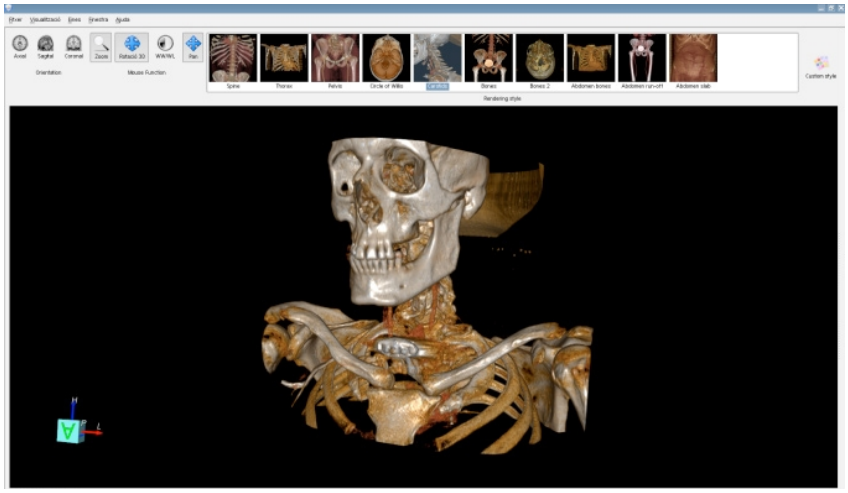


Figura 2.1.2.1: captura de l'eina STARVIEWER

2.1.3 ITK-SNAP

Pàgina web: http://www.itksnap.org/pmwiki/pmwiki.php

SNAP és una aplicació software utilitzada per segmentar estructures en imatges mèdiques 3D.

Proporciona eines per navegar a través de la imatge. També ofereix la possibilitat de enllaçar múltiples imatges i mostrar-les concurrentment.

Suporta una gran varietat de tipus d'imatges 3D.

Una cosa curiosa d'aquesta aplicació és que també utilitza la llibreria ITK, la mateixa que nosaltres ens proposem utilitzar.

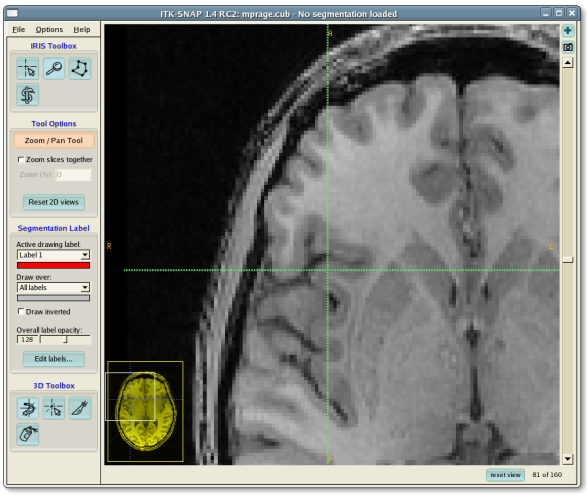


Figura 2.1.3.1: Ampliació d'una àrea d'un tall d'una imatge, feta amb SNAP-ITK

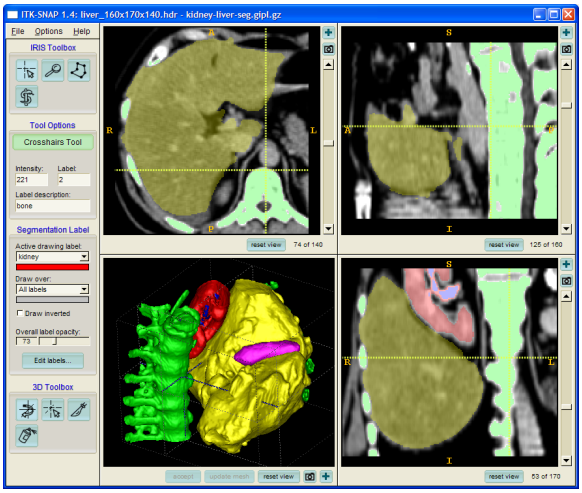


Figura 2.1.3.2: Navegació per imatges que mostren els ronyons i el fetge

2.1.4 VueMe

Pàgina web: http://www.mimsoftware.com/products/vueme/

Aplicació per dispositius iOS desenvolupada per l'empresa MIM Software.

Dissenyada pels pacients, perquè puguin veure les imatges enviades pels seus metges. Si cal, les haurien de veure amb un especialista.

Utilitza un sistema anomenat MIMCloud, que el que fa es tenir les imatges guardades remotament, i llavors accedir-hi des de l'aplicació a través d'internet. Quan una imatge és obtinguda, es transmet xifrada, per tal d'assegurar al màxim la privacitat del pacient.

Alguns exemples de les capacitats d'aquesta aplicació són:

Figura 2.1.4.1: exemple d'imatges visualitzades amb VueMe

Figura 2.1.4.2: imatge d'un tumor mostrat utilitzant VueMe

2.1.5 Chili Digital Radiology

Pàgina web: http://www.chili-radiology.com/en/

Aplicació per dispositius mòbils.

Les funcionalitats més importants que inclou són:

- Independència del sistema operatiu (iOS, Android, etc.), ja que es basa amb un visualitzador basat en web (utilitzant HTML5). Més que una aplicació, és una web.
- Té funcions com zoom, pan, anotacions, etc.
- Xifrat de les imatges.
- Alta velocitat de comunicació gràcies a compressió de les dades transmeses per WLAN o UMTS.

Figura 2.1.5.1: Chili en un iPad

Així doncs, com podem veure, hi ha vàries eines ja existents[6] que fan gran part del que nosaltres ens proposem, i fins i tot més. Algunes d'elles ja són utilitzades des de dispositius mòbils i tot.

Queda una última cosa per mencionar com a antecedent:

2.1.6 ITK a iOS

Que és possible incloure la llibreria ITK a iOS està demostrat per l'existència de la guia *ITK on iOS*.

Que sigui possible utilitzar-la per representar les imatges mèdiques en format 3D està per veure. Ara bé, existeix una continuació d'aquesta guia, anomenada *ITK Image IO interface with Apple iOS*. Aquesta segona guia l'únic que ensenya és a utilitzar la llibreria ITK amb algun dels seus filtres, a mostrar-la en 2D per la pantalla del dispositiu, i a guardar el resultat de la imatge (després d'aplicar algun filtre) a la biblioteca d'imatges del dispositiu.

[6] Més eines a la referència extra [2]

2.2 Factibilitat

Un dels factors que pot limitar el projecte, és la capacitat dels dispositius mòbils que utilitzin iOS per córrer l'aplicació que es vol desenvolupar.

Tot i que en principi no hauria d'haver problema, vistos els antecedents, a continuació s'analitzaran les característiques dels últims models d'aquests dispositius (presents al mercat en el moment de començar el projecte), ja que són considerats els dispositius objectiu. Si a més l'aplicació pogués ser executada en versions més antigues dels mateixos dispositius, seria considerat un valor afegit extra.

2.2.1 iPod Touch 4G

Comencem pel que serà el dispositiu físic final amb el que es faran les proves.
Les característiques que ens interessen d'aquest dispositiu són les següents:

Hardware	Xip	Apple A4
	Processador	Single core
	Disponibilitat de processador gràfic	Sí
	RAM	256MB
	Capacitat d'emmagatzematge	8/16/32GB
Software	Sistema Operatiu	iOS 5.0
Sensors	Giroscopi	Giroscopi en 3 eixos
	Acceleròmetre	Sí

En un primer anàlisi ràpid podem concloure que espai no en faltarà. Per molt grans que puguin ser les imatges mèdiques, amb 8GB, com a mínim, de disponibilitat no hauria d'haver problemes.
Per altra banda, 256MB de RAM són forces menys que qualsevol ordinador dels actuals, però ja és ben sabut que en un dispositiu mòbil la memòria és un dels punts claus a tenir en compte a l'hora de programar. S'haurà d'intentar optimitzar l'ús de la memòria, però tampoc sembla que hagi de donar problemes.

Figura 2.2.1.1: iPod Touch 4G

A més, l'iPod disposa de giroscopi, el que permetrà fer que la imatge representada es pugui moure no només amb gestos tàctils sinó també amb la inclinació del propi dispositiu.

2.2.2 iPhone 4/4S

Tot i que sembla que hauria de ser molt semblant a l'iPod Touch, anem a veure quines característiques té un iPhone 4, el disponible en el moment de començar el projecte:

Hardware	Xip	Apple A4/A5(iPhone 4S)
	Processador	Single core
	Disponibilitat de processador gràfic	Sí
	RAM	512MB
	Capacitat d'emmagatzematge	8/16/32GB
Software	Sistema Operatiu	iOS 5.0
Sensors	Giroscopi	Giroscopi en 3 eixos
	Acceleròmetre	Sí

L'única diferència és que l'iPhone encara disposa de més memòria RAM que l'iPod, amb el que l'aplicació hauria de poder-s'hi executar sense problemes, si es pot executar en l'iPod.

Figura 2.2.2.1: iPhone 4S

2.2.3 iPad 2

La segona versió d'iPad també estava disponible en el moment de començar el projecte. Aquestes són les seves característiques rellevants:

Hardware	Xip	Apple A5
	Processador	Dual core
	Disponibilitat de processador gràfic	Sí
	RAM	512MB
	Capacitat d'emmagatzematge	16/32/64GB
Software	Sistema Operatiu	iOS 5.0
Sensors	Giroscopi	Giroscopi en 3 eixos
	Acceleròmetre	Sí

A part de l'augment en capacitat d'emmagatzematge, disposa de la mateixa RAM que l'iPhone 4, i el mateix xip que l'iPhone 4S, pel que no hi hauria d'haver gaire diferència entre una execució de l'aplicació en un o altre dispositiu.

Figura 2.2.3.1: iPad 2 versió en negre i versió en blanc

2.2.4 Dispositius llençats posteriorment

La tendència d'Apple és llençar pràcticament anualment una nova versió dels seus dispositius cada any. Així doncs, en el transcurs del projecte[7], has sorgit nous dispositius els quals no s'analitzen però que, és d'esperar, tindran millors característiques que les seves versions anteriors i que, per tant, haurien de poder executar l'aplicació objectiu d'aquest projecte, i l'haurien de poder executar en millors condicions.

Concretament, els dispositius que han sortit al mercat durant el període de fer el projecte són:

- iPad 3
- iPhone 5
- iPod Touch 5G
- iPad Mini
- iPad 4

Sembla ser doncs que els dispositius no haurien de suposar un problema alhora de poder fer l'aplicació que es demana, sempre i quan es faci un bon ús de la memòria i s'evitin les fugues de memòria.

2.2.5 Factor humà

Un últim factor a tenir en compte és el factor humà. L'aplicació requerirà programar per iOS utilitzant un llenguatge anomenat Objective-C, el qual es desconeix completament.

A més, caldrà afegir una llibreria externa, que es desconeix, i se'n sap ben poca cosa.

Per últim, també exigirà utilitzar OpenGL, del qual no es tenen sinó unes bases.

Així doncs, tot i que pot sonar motivador el fet d'aprendre sobre noves tecnologies (o augmentar el coneixement d'algunes ja existents), cal tenir en compte que això és un factor que pot influir negativament en el desenvolupament del projecte.

En principi, però, s'espera assolir el nivell requerit per l'aplicació.

---

[7] Cal destacar que per *transcurs del projecte* s'entén el temps real que s'ha consumit en el projecte, no el previst en la planificació inicial. Més detalls en la secció 6.

3. Desenvolupament tècnic
Abans de començar, cal fer un apunt:

Malauradament i com es justificarà poc a poc a continuació, aquest projecte no es va completar amb èxit.

Per aquesta raó, aquest apartat de la memòria i els següents tindran un format més aviat de línia temporal, en comptes del format més típic en el qual es dissenya l'aplicació i posteriorment s'implementa.

Primerament, s'explica tot allò que es va aconseguir exitosament tot i les traves que van anar apareixent.

Posteriorment, s'expliquen les diferents opcions explorades i provades abans d'abandonar l'intent de finalitzar el projecte, ja que es feia necessari redactar aquest mateix document. Es va prendre aquesta decisió ja que, si no es podia acabar el que s'havia previst, com a mínim s'havia d'aconseguir plasmar en aquest document tota la feina realitzada. D'altra banda tindríem un codi a mig fer per una banda i una memòria feta ràpida i malament a l'altra. Més detalls a la secció 6.

3.1 Arrencada inicial
Abans de començar a dissenyar la que seria l'aplicació final, el que es va fer va ser intentar conèixer l'entorn on es desenvoluparia el projecte, l'Xcode, tot jugant amb petits tutorials que poguessin ensenyar com funciona.

Alguns exemples de tutorials seguits són la referències extres que van de la [3] a la [9].

Posteriorment, ja es va començar a mirar com s'havia de fer l'aplicació final, amb uns primers esbossos del que en seria el disseny.

Abans de descriure'ls, però, parlem de dos dels elements bàsics: el llenguatge amb el que es programarà l'aplicació, l'Objective-C, i la llibreria ITK.

3.1.1 Objective-C
És el llenguatge d'alt nivell que utilitza tant qualsevol dispositiu OSX (Macbook, iMac, Mac Mini...) com qualsevol dispositiu dels mòbils (iPhone, iPod, iPad).

L'Objective-C és un superconjunt del C, és orientat a objectes i té una sintaxi de missatges semblant a Smalltalk. Aquests missatges estan composats per un destinatari i una acció a realitzar.

Està pensat per utilitzar el conegut patró de disseny MVC: Model Vista Controlador.

Per fer un repàs ràpid d'aquest patró:

- Els models són objectes, instàncies de classes, que mantenen la informació bàsica de l'aplicació. Contenen les dades que tendeixen a necessitar persistència, i defineixen la lògica per manipular aquestes. Idealment, estan totalment aïllades de la interfície d'usuari i són per tant força reutilitzables, ja que representen un concepte o idea.
- Les vistes són les que presenten la informació a l'usuari i interaccionen amb ell. No emmagatzemen les dades que representen, només pot tenir algun tipus de memòria cau per temes d'optimització. Una vista pot representar una part d'un objecte del model, o un objecte sencer o fins i tot varis objectes del model.
- Els controladors actuen com a intermediaris entre les vistes i el model. Una vista ha de ser notificada quan alguna cosa ha canviat en el model, per poder-ho presentar a l'usuari. De la mateixa manera, una vista que rep algun tipus d'interacció amb l'usuari, normalment ha de provocar canvis en alguna part del model. D'aquest tipus de comunicació és del que s'encarreguen els controladors.

Tot i que Objective-C està pensat per utilitzar aquest patró, ofereix una mica més de flexibilitat. Per això, conceptualment defineix dues classes de controladors:

- Els *view controllers* són aquells que bàsicament es relacionen amb vistes. "Posseeixen" les vistes, i la seva responsabilitat principal és la de manejar la interfície i comunicar-se amb el model. Mètodes que tenen a veure amb les dades mostrades són típics d'aquesta classe de controladors.
- Els *model controllers* són aquells que bàsicament estan lligats amb la capa del model. "Posseeixen" el model. Les seves responsabilitats són de maneig del model i de comunicar-se amb les vistes. Mètodes que cal aplicar al model en conjunt són típicament implementats en aquest tipus de controladors.

Per altra banda, el propi Cocoa[8] defineix dos tipus de controladors genèrics:

- Els *mediating controllers* són els típics controladors, els que faciliten el flux de dades des de les vistes fins els models, i a la inversa. Són els controladors que poden ser inclosos utilitzant l'*Interface Builder* de l'Xcode. Es poden relacionar els objectes de la vista amb propietats d'aquest controlador. Alhora, aquestes propietats es poden lligar amb propietats d'un objecte del model. Així es comuniquen els canvis en qualsevol de les dues parts.

[8] Cocoa és un entorn de desenvolupament format per un conjunt de frameworks, que estan formats per llibreries i APIs, i que formen la capa de desenvolupament per tots els sistemes OS X i iOS.

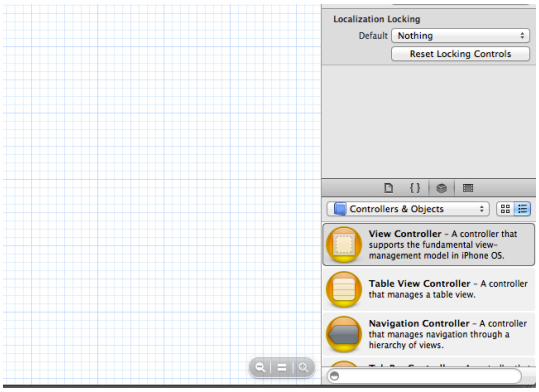


Figura 3.1.1.1: Part de l'Interface Builder amb el menú amb mediating controllers.

- Els *coordinating controllers* són els encarregats de supervisar i coordinar el funcionament de tota l'aplicació (o una part d'ella). Tenen responsabilitats tals com:
  - Respondre a missatges delegats i observar notificacions.
  - Respondre a missatges d'acció[9].
  - Manejar el cicle vital dels objectes que posseeix, per exemple, alliberant els recursos que ocupen quan pot. Aquests objectes poden ser perfectament *mediating controllers*.
  - Establir connexions entre diferents elements i altres tasques típiques d'una inicialització.

La figura següent mostra la relació ideal entre els dos tipus de controladors. Els arxius "nib" són els que contenen la definició de la interfície gràfica, construïts usant l'*Interface Builder*.

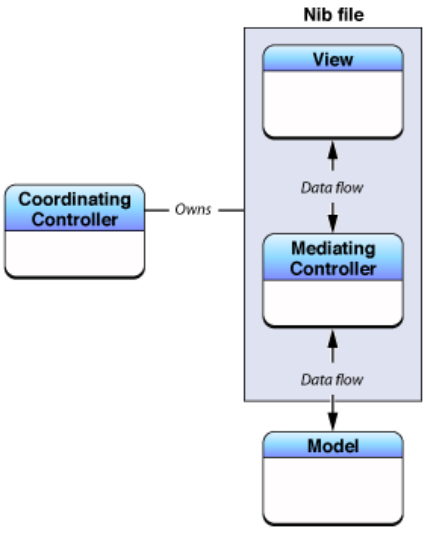


Figura 3.1.1.2: Relació entre coordinating controllers i mediating controllers.

[9] Un missatge d'acció o *action message* és aquell missatge enviat quan un event es dispara. El típic senyal que s'envia, per exemple, quan un botó es prem.

Com que originalment no es coneixia absolutament res el llenguatge, per començar es volia buscar algun tipus de tutorials per introduir-s'hi. La gran sort va ser trobar un curs complet de vídeos a l'iTunes U[10] de la universitat d'Stanford que ensenyava una introducció al llenguatge.

Concretament, les col·leccions de vídeos mirades (parcialment) són:

- iPad and iPhone Application Development
- Developing Apps for iOS

Si bé òbviament no es podien fer preguntes, la sensació de ser a classe es produïa.

A més, es tenia la oportunitat de rebobinar en cas necessari.

Malauradament, però, va demostrar no ser un mètode molt eficient, ja que els vídeos tenien una duració mitjana d'una hora i vint minuts, mentre que si estigués tot narrat en un document probablement no es necessités tant de temps per llegir-lo i comprendre'l.

3.1.2 Llibreria ITK

3.1.2.1 Descripció

La llibreria ITK o *Insight Toolkit* es una llibreria de codi obert la funció de la qual és el registre i la segmentació.

Per **segmentació** s'entén el procés d'identificar i classificar informació obtinguda a través de mostres i emmagatzemada en forma digital. Típicament, aquestes imatges són les obtingudes per escàners CT[11] o MRI[12].

En canvi, el **registre** és la tasca de trobar correspondències entre les dades. Per exemple, es pot utilitzar el resultat d'un escàner CT i un MRI per combinar la informació continguda en ambdós i poder tenir una visió més àmplia.

La llibreria ITK està programada en C++ i és multi plataforma. Es construeix al voltant de l'entorn CMake, que s'encarrega de compilar de manera independent de la plataforma.

Utilitza *templates* per intentar programar de forma genèrica, és a dir, que el mateix codi es pugui executar independentment del tipus de les dades. Les templates s'utilitzen generalment alhora que la sobrecàrrega d'operadors (*operator overloading*).

---

[10] Secció de l'aplicació iTunes Store on els usuaris es poden descarregar cursos educatius, provinents de diferents universitats, de forma totalment gratuïta. Els cursos són normalment en format àudio o vídeo.

[11] *Computed Tomography*: és un procés d'obtenció d'imatges a partir de Rajos X processats per ordinador, amb l'objectiu de produir imatges tomogràfiques (o "talls") de certes parts del cos.

[12] *Magnetic Resonance Imaging*: tècnica utilitzada en la radiologia per visualitzar les estructures internes del cos en detall. Utilitza ressonància magnètica nuclear (NMR) per agafar mostres dels nuclis dels àtoms de dins del cos. Amb MRI s'obtenen imatges del cos molt més precises que amb Rajos X.

A més, l'ús de templates fa que l'execució del codi sigui altament eficient, i que molts problemes en el codi es puguin detectar en temps de compilació, sense haver d'arribar a execució.

3.1.2.2 Incloent la llibreria

Per incloure la llibreria cal utilitzar CMake, tal com havíem dit anteriorment. Utilitzant la guia *ITK on the iOS* es va incloure la llibreria a l'Xcode.

Durant la inclusió de la llibreria van sorgir petits problemes de compatibilitat, degut a que la versió d'Xcode a la que es fa referència la guia és més antiga, però es van poder solucionar amb un temps raonable simplement veient els errors que el propi Xcode donava i actuant en conseqüència.

Ara bé, el fet d'incloure la llibreria va provocar un primer problema:

3.1.2.3 Primer problema

En el moment que la llibreria ITK va ser inclosa exitosament en el projecte a l'Xcode, **es va perdre la possibilitat d'executar l'aplicació en el simulador**.

Mentre que l'aplicació s'executava sense problemes en l'iPod, al intentar fer córrer l'aplicació en el simulador d'iPhone 5 es generaven errors de linkatge. Errors que, després d'una recerca exhaustiva per internet, no es van poder solucionar. El màxim que es va trobar va ser posts a fòrums amb gent amb problemes similars, la majoria sense resposta.

Tot i així, tampoc se li va donar gran importància, degut a què es va pensar que si l'aplicació podia córrer en el dispositiu físic, no hi havia necessitat de recórrer més al simulador.

En el moment de començar el projecte, feia un mes acabava de sortir una nova actualització important de la llibreria cap a la versió 4.0, però per desgracia no acabava de ser compatible amb la guia que es basava en la versió 3.2, ja que els canvis eren massa significatius com perquè fos quelcom immediat (i no es volia perdre excessiu temps en intentar adaptar-la. Es va provar però no semblava fàcil).

Així doncs, la versió utilitzada és la versió 3.2. Això no s'ha de veure com a quelcom negatiu, tota llibreria acabada de sortir és més propensa a contenir errors o *bugs*, així que tot i perdre un nombre considerable de funcionalitats afegides en la nova versió, tampoc va ser una mala passada haver d'utilitzar la versió anterior 3.2.

3.1.3 OpenGL vs VTK

Investigant la llibreria ITK, es va descobrir que la majoria d'aplicacions que la utilitzaven també utilitzaven una altra llibreria, la llibreria VTK.

La llibreria VTK, o *Visualization Toolkit*, és una llibreria per processar i representar imatges 3D. Programada en C++, però també té interfícies per poder ser utilitzada conjuntament amb llenguatges com Java o Python.

Té un gran conjunt d'algorismes de visualització, que inclouen escalars, vectors, textures i mètodes volumètrics.

També disposa d'avançades tècniques de modelització com són la reducció de polígons, la suavització de malles i contorns.

És multiplataforma. Està disponible per Windwos, Mac OSX i Linux.

Així doncs, perquè no escollir VTK per a representar les imatges mèdiques?

En primer lloc, perquè incloent ja una llibreria com és ITK, la mida total de l'aplicació ja creix molt, i incloure una altra llibreria podria augmentar molt la mida de l'executable final. Tot i que això no sigui un problema directament, si la intenció és que l'aplicació s'utilitzi, i aquesta ocupa molt, els usuaris no la voldran mantenir en els seus dispositius: serà de els primeres a esborrar quan necessitin espai.

Però aquest no és el problema principal. El problema principal és que no es disposava d'un mètode immediat d'incloure la llibreria. Igual que ITK, VTK està pensat per aplicacions d'escriptori. Si incloure la llibreria ITK ja havia donat certs problemes tot i disposar d'una guia, d'VTK no se'n sabia res. De fet, formava part de l'objectiu del projecte la representació de la imatge en 3D. D'aconseguir adaptar VTK a iOS, el projecte es podria quedar curt en objectius, ja que hagués estat la simple inclusió de dues llibreries i d'interaccionar entre elles.

El problema que suposava haver decidit utilitzar OpenGL és que pràcticament tota la documentació, posts a fòrums, etc. que indicaven com representar les imatges mèdiques, utilitzaven VTK. No hi havia res que indiqués com fer-ho amb OpenGL directament. De fet, la pròpia pàgina de la wiki d'ITK referent als mètodes de visualització[13] estava (i està encara) incompleta.

Per tot això, es va acabar decidint utilitzar OpenGL directament per a la representació de la imatge en 3 dimensions.

[13] http://www.itk.org/Wiki/ITK/Visualization_Toolkits

3.2 Aplicació final - Visualitzador mèdic

3.2.1 Disseny inicial de l'aplicació

A continuació es presenta el que seria el disseny de l'estructura inicial de l'aplicació. És una estructura inicial en el sentit que inclou la interfície d'usuari mínima, directament formada pels elements necessaris per fer funcionar l'aplicació, obviant per exemple, un menú principal ben definit, amb elements de guia.

No s'inclouen ni atributs ni operacions ja que es desconeixia el funcionament d'Objective-C i els seus tipus de dades el suficient com per incloure-ho en el disseny inicial, però tampoc pel que ve a ser l'estructura ja és suficient.

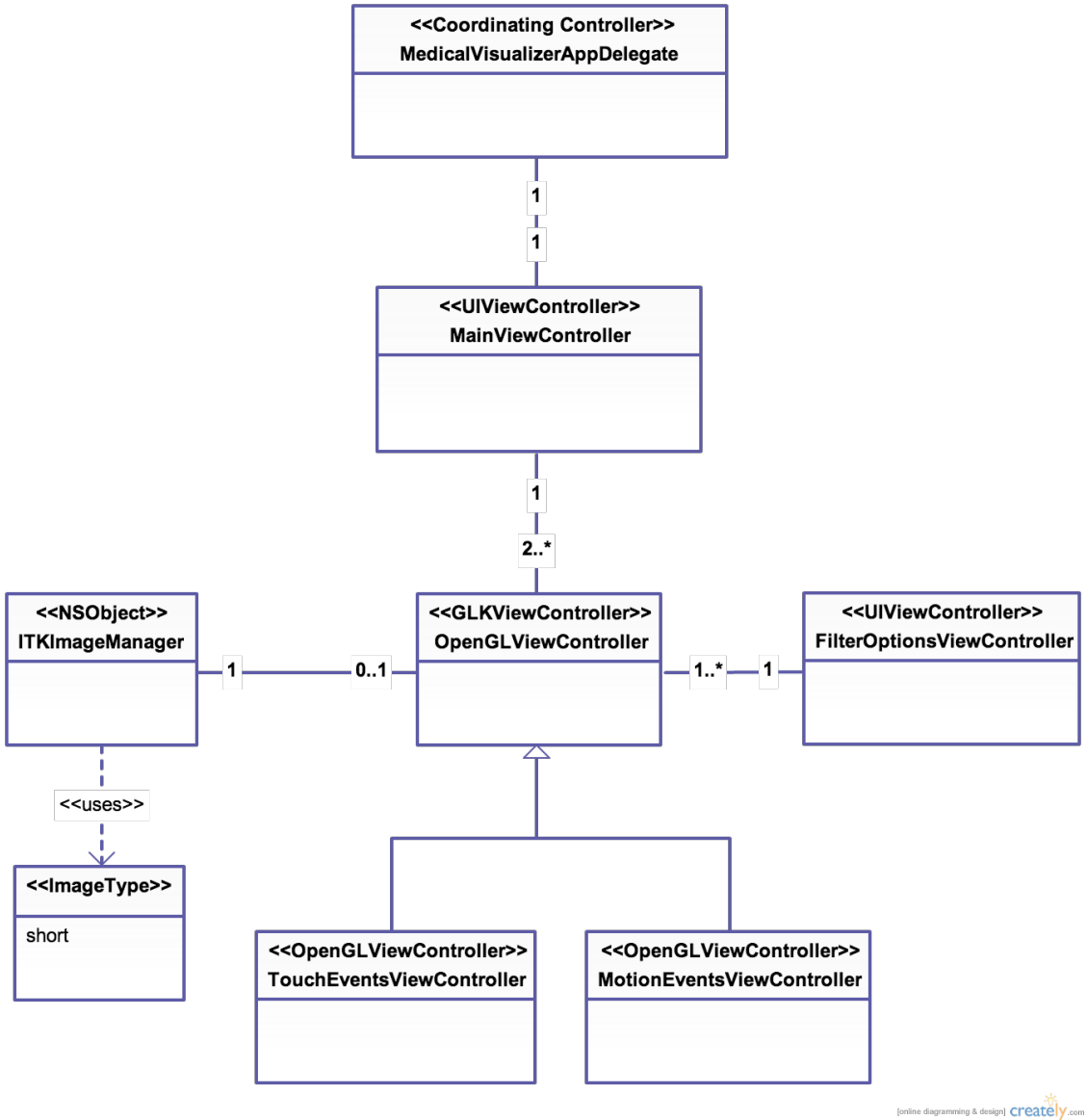


Figura 3.2.1.1: Diagrama UML de l'estructura de l'aplicació.

Les responsabilitats de cada component descrites a continuació:

3.2.1.1 MedicalVisualizerAppDelegate

Aquest seria el controlador principal de l'aplicació. Segons la classificació de controladors vista a l'apartat 3.1.1, equivaldria al *coordinating controller* de l'aplicació. Seria l'encarregat de coordinar, com el nom diu, el funcionament de tota l'aplicació. Tindria la responsabilitat d'alliberar els recursos que fos possible en el moment que l'aplicació sortís del pla principal, per exemple perquè l'usuari prem el botó *home* del dispositiu o el d'encès/apagat.

3.2.1.2 MainViewController

Com el nom indica, seria el *view controller* principal. Seria d'un tipus anomenat *navigation controller*. Aquest tipus de controlador consta de:

- Una vista amb una taula que conté diferents opcions, on cadascuna carregarà una nova vista apilant-la damunt l'actual.
- Una *navigation bar* (barra de navegació) situada a la part superior de la vista, que conté el títol que s'hagi assignat a la vista actual i el botó per tornar a l'anterior.
- Opcionalment, una *navigation toolbar*, una barra situada a la part inferior de la vista, on s'hi poden afegir botons que tinguin funcionalitats vàries.

Figura 3.2.1.2.1: Exemple de navegació utilitzant un navigation controller. En la imatge es pot veure la navigation bar, però no s'utilitza navigation toolbar.

La idea és utilitzar aquest controlador per mostrar una vista que permeti escollir entre dues opcions: si utilitzar els gestos tàctils per moure la càmera dins l'escena o si utilitzar el giroscopi. Podria ser que finalment es poguessin ajuntar ambdues funcionalitats en una, però inicialment es considera millor mantenir-ho separat.

També sembla que hagi de ser millor de cara a l'experiència d'usuari només utilitzar gestos o només giroscopi, d'altra banda podria provocar-li problemes per orientar l'escena.

3.2.1.3 OpenGLViewController

Aquest controlador seria l'encarregat d'agrupar tota la part de funcionalitats d'OpenGL. És del tipus *GLKViewController* perquè aquest és el tipus dels controladors que implementen el bucle de renderització d'OpenGL ES. Alguns dels

seus mètodes són els que cal implementar per tal de poder dibuixar una escena OpenGL.

El que es té pensat és utilitzar la *navigation toolbar*, que proporcionaria el *navigation controller* pare, per incloure-hi un botó que sigui l'encarregat d'avisar la instància de *FilterOptionsViewController* de que ha de mostrar la vista per escollir les diferents opcions de filtres.

3.2.1.4 TouchEventsViewController

Aquest controlador seria una extensió del controlador *OpenGLViewController* que, a més d'implementar les funcionalitats del pare (a través de la simple herència), implementaria el necessari per afegir la capacitat d'editar la càmera de l'escena utilitzant els gestos tàctils.

3.2.1.5 MotionEventsViewController

El *MotionEventsViewController* seria una altra extensió del tipus de controlador *OpenGLViewController* que, a més de les funcionalitats del pare, implementaria tot allò necessari per permetre moure la càmera de l'escena utilitzant el giroscopi.

3.2.1.6 ITKImageManager

Tot i que no és necessària la persistència en el tipus d'aplicació que estem a punt de desenvolupar, si hi ha alguna cosa que pugui ser considerada "model" (dins de la separació de components model-vista-controlador) és aquesta classe.

Aquesta classe seria l'encarregada de dur a terme totes les operacions relacionades amb la llibreria ITK, començant per llegir la imatge del fitxer i acabant per aplicar els filtres que es volguessin aplicar.

La definició d'aquesta classe no només permet encapsular les funcionalitats d'ITK en un sol component, sinó també aïllar al màxim els dos tipus de llenguatges diferents: C++ de la llibreria i Objective-C de la resta de l'aplicació. Això proporciona un avantatge clar, ja que entremesclar massa els dos llenguatges podria comportar més risc d'errors. D'aquesta manera, tot el codi C++ queda aïllat i la detecció de *bugs* causats per aquest hauria de ser més fàcil.

3.2.1.7 FilterOptionsViewController

Finalment, aquest controlador seria l'encarregat d'obrir una nova vista on s'escollirien les diferents opcions de filtres a aplicar: quins activar, quins desactivar, els paràmetres que fossin necessaris, etc.

3.2.2 Lectura de la imatge ITK

El format d'imatges mèdiques comú és el de *Meta-Image*, que està format per dos fitxers, un d'ells és una capçalera amb la informació d'alguns paràmetres de la imatge i l'altre és la imatge en sí.

En el cas de la imatge mèdica utilitzada en aquest projecte, la imatge utilitzada consta dels dos fitxers mencionats:

- Un fitxer amb extensió *mha* que conté els paràmetres de la imatge tals com:
- Tipus de l'objecte: imatge.
- Nombre de dimensions de la imatge: 3.
- Nombre d'elements per dimensió: 256 en l'eix X, 256 en l'eix Y i 139 en l'eix Z.
- Tipus dels elements: MET_SHORT (equivalent a short, 2 bytes per representar un punt).
- Com estan la informació guardada a memòria: utilitzant LSB[14] (little-endian).
- Espai entre els voxels[15] en els 3 eixos: 0.976562 en l'eix X i Y i 1.0 en l'eix Z.
- Finalment, un punter al fitxer *raw* que conté la imatge pròpiament.
- La imatge en sí, com a seqüència de bytes, emmagatzemada en un fitxer d'extensió *raw*.

La imatge està titulada amb l'original nom d'*image01*, amb el que els fitxers s'anomenen *image01.mha* i *image01.raw*.

Per llegir la imatge, a partir de la guia *ITK on the iOS*, es va extreure el codi necessari.

Inicialment es va intentar llegir la imatge utilitzant les 3 components de color (R, G i B) però no tenia sentit, donat que la imatge estava representada com a un valor de luminància a cada píxel donat (representat pel short), pel que al llegir amb els 3 components es replicava la informació 3 vegades, cosa que omplia molta més memòria. Per tant, es va llegir la imatge directament com a escala de grisos.

```
typedef short pixelType; //Defineix el tipus d'un pixel
```

El següent codi mostra com es fa per llegir la imatge, i serveix també per veure un exemple de codi de la llibreria ITK:

```
typedef itk::Image<pixelType, 3> imageType; //Una imatge de 3
dimensions
imageType::Pointer ITKImage; //Contindrà la imatge després de
llegir
typedef itk::ImageFileReader<imageType> LuminanceReaderType;
//Tipus del lector
```

[14] Mecanisme de guardat de les dades en memòria en que el byte menys significant es guarda primer. Exemple: 05F5E100 es guardaria a memòria 00 E1 F5 05.
[15] Voxel o *volumetric pixel* un element de volum. Representa un valor en una graella en espai tridimensional. Cada punt de la imatge ITK equival a un voxel.

```
LuminanceReaderType::Pointer LuminanceReader =
LuminanceReaderType::New(); //Instanciació del lector
NSString * pathToImage = [[NSBundle mainBundle]
pathForResource:@"image01" ofType:"@mha" inDirectory:nil];
//Codi Objective-C per buscar el path de la imatge dins els
recursos de l'aplicació
LuminanceReader->SetFileName(pathToImage.UTF8String); //D'on
llegir la imatge
LuminanceReader->Update(); //Sense aquesta instrucció la lectura
no s'executa
ITKImage = LuminanceReader->GetOutput(); //S'obté la imatge
llegida
```

Com es pot observar, el codi és força intuïtiu, amb uns noms de funcions molt ben definits, amb els que resulta força entenedor. Tot i així conté parts que a priori poden semblar estranyes, com la definició d'un tipus pel lector. Això és degut a l'ús de *templates*, anteriorment anomenats, per tal de fer que el codi sigui el màxim de reaprofitable, sigui quin sigui el tipus de les dades.

Al aplicar-lo, va sorgir un dels primers problemes: la imatge era molt gran i causava un *Memory Warning*.

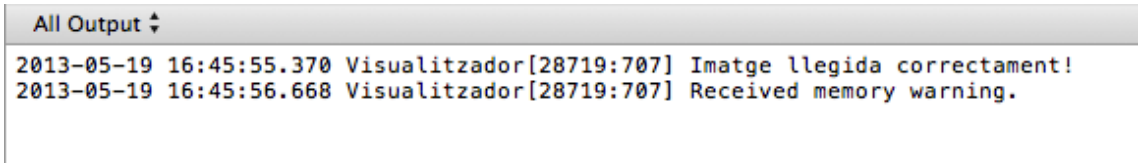


Figura 3.2.2.1: missatge de memory warning.

La imatge ocupa:

$$\#elements\ en\ eix\ X * \#elements\ en\ eix\ Y * \#elements\ en\ eix\ Z * mida\ d'un\ element$$

És a dir:

$$256 * 256 * 139 * 2 = 18.219.008\ Bytes \approx 18\ Megabytes$$

18 Megabytes només tenir la imatge llegida a memòria!

Per a comprovar que efectivament ocupés això, es va utilitzar l'eina "Instruments" de Mac OS X. Entre altres coses, permet veure la memòria utilitzada a cada moment (mentre s'executa l'aplicació). La sortida ens ho confirmava:

Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes	# Overall	# Allocations (Net / Overall)
☑	* All Allocations *	19,09 MB	24491	0	19,09 MB	24491	+++
☐	Malloc 17,38 MB	17,38 MB	1	0	17,38 MB	1	

Figura 3.2.2.2: Sortida de l'aplicació Instruments que confirma la mida de la imatge (la mida és inferior a la calculada ja que divideix per 1024^2 els bytes).

Bé, i és important que surti aquest missatge? Ens pot afectar substancialment?

Doncs sí, un cop rebut aquest missatge, la memòria disponible abans que l'aplicació rebi un *signal*[16] del propi sistema operatiu per tancar-se no és excessiva (ens hi vam trobar).

La manera de solucionar aquest problema va ser utilitzar el tipus *unsigned char* com a tipus de píxel de la imatge, de manera que el lector ITK el transformés, i així ocuparia només la meitat de l'espai. El perquè *unsigned char* i no simplement *char* serà explicat més endavant. L'únic canvi necessari en el codi anterior és doncs la línia:

```
typedef short pixelType;
```

per:

```
typedef unsigned char pixelType;
```

Amb això, el *memory warning* desapareix de la consola en el moment de l'execució.

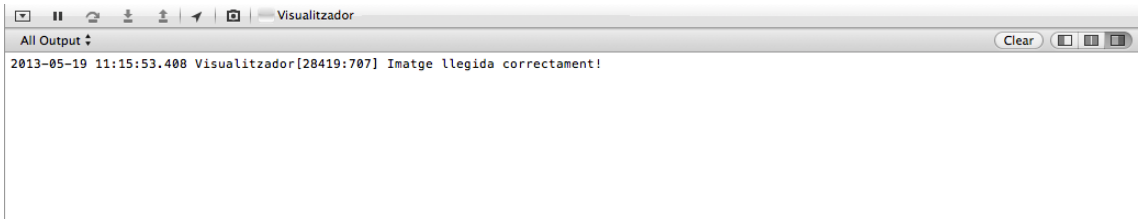


Figura 3.2.2.3: Consola de l'Xcode sense memory warning.

Si mirem ara la sortida d'Instruments:

Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes	# Overall	# Allocations (Net / Overall)
☑	* All Allocations *	10,40 MB	24500	0	10,40 MB	24500	+++
☐	Malloc 8,69 MB	8,69 MB	1	0	8,69 MB	1	

Figura 3.2.2.4: Sortida de l'aplicació Instruments que confirma que la mida s'ha reduït a la meitat.

3.2.2.1 Comprovació de que la imatge s'ha llegit correctament

Utilitzant la continuació de la guia *ITK on the iOS*, *ITK Image IO interface with Apple iOS*, es va poder comprovar que la imatge es llegia correctament. Per desgràcia, com ja s'havia dit, en aquesta segona guia només es parla de l'escriptura de les imatges en la galeria d'imatges del dispositiu, i imatges en dues dimensions. És una bona manera de comprovar si el lector proposat funciona, però no servirà pel futur del projecte, on es busca la representació en tres dimensions.

Cal dir que abans de fer proves amb la imatge mèdica es van fer proves amb altres imatges en altres formats (png, jpeg) ja que pràcticament es llegeixen d'igual manera, i així es podien separar més les proves. Exemples:

[16] Notificació asíncrona enviada a un determinat procés, per informar d'un event ocorregut. En aquest cas, si l'aplicació ha sobrepassat la memòria màxima que se li cedeix, se li envia un *signal* donant l'ordre de tancar-se.


Figures 3.2.2.1.1 i 3.2.2.1.2: logo d'Apple llegit correctament i imprès abans i després d'aplicar un dels filtres disponibles.

Figura 3.2.2.1.3: Un tall de la imatge mèdica.

Així doncs aquesta era la primera vegada que aconseguíem mostrar (part de) la imatge ITK llegida, *ergo* l'estàvem llegint correctament.

3.2.3 Primers intents de representar la imatge en 3D

A continuació es descriuen els passos seguits i totes les alternatives explorades, a l'hora d'intentar representar la imatge en tres dimensions, punt que era clau en l'avenç del projecte.

3.2.3.1 Un punt per píxel

La llibreria ITK proporciona un conjunt d'iteradors per poder iterar pels píxels corresponents a alguna de les dimensions de la imatge. Aquests iteradors es poden utilitzar tant per llegir la informació d'un píxel determinat com per aplicar una determinada funció (filtre) per cada píxel aprofitant les iteracions.

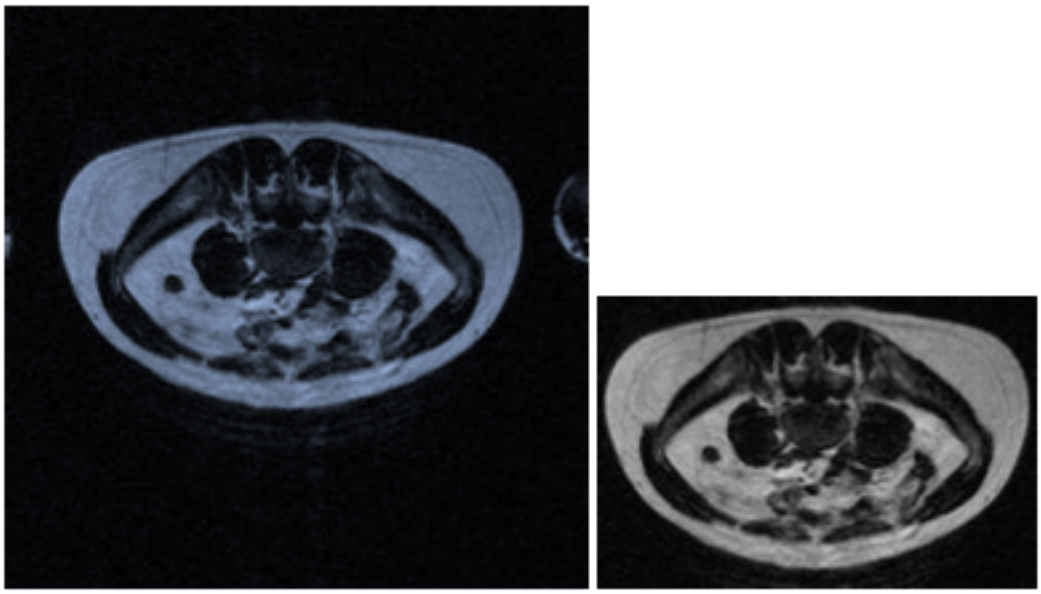


Figura 3.2.3.1.1 A l'esquerra, imatge original. A la dreta, resultat d'aplicar l'iterador ImageRegionIteratorWithIndex per retallar part sobrant de la imatge (cropping).

La primera ocurrència de com representar la imatge era, doncs, utilitzar aquests iteradors per recórrer la imatge en els tres eixos i utilitzar la funció d'OpenGL *glBegin*, que permet definir primitives[17] geomètriques durant el procés de renderitzat.

Aquesta funció permet definir els vèrtexs "a l'aire" utilitzant, per exemple, la funció *glVertex3f* (aquesta funció especifica les coordenades d'un vèrtex). En acabat, quan un cert polígon que es vol definir s'acaba, es tanca la definició de vèrtexs utilitzant *glEnd*.

[17] Una primitiva geomètrica és l'objecte geomètric bàsic amb el que es formen els polígons a pintar. Pot ser des d'un simple punt, format per un vèrtex, a per exemple un quadrilàter, format per 4 vèrtexs.

La idea era doncs utilitzar *glBegin* amb el tipus de primitiva *GL_POINTS*[18] i separant cada vèrtex de l'anterior un valor *n* en l'eix corresponent (per fer que els punts formin la imatge). Aquest valor s'hagués buscat empíricament.

Podria ser un mètode massa ineficient, degut a haver de recórrer la imatge sencera, cosa que possiblement no fes falta si es tinguessin en compte solapaments i punts no visibles de la imatge.

A més, el fet de cridar la definició dels vèrtexs en la mateixa funció de renderitzat seria extremadament ineficient. Si a més el nombre de vèrtexs a definir fos elevat, podria ser inviable.

Això no seria un problema però. Existeixen les anomenades *display lists*, que el que permeten aquestes és encapsular la definició d'un polígon entre les crides *glBegin* i *glEnd* tot assignant-li un número identificador de *display list*. Posteriorment, simplement utilitzant *glCallList(NumeroDisplayList);* es pot pintar el polígon, havent definit únicament un cop els vèrtexs.

Així doncs, l'eficiència es podria aconseguir amb aquestes llistes.

Una altra cosa que podria passar és que la mida dels punts no fos suficient per representar bé la imatge (podrien ser massa petits i deixar massa espai entre ells, amb el que quedaria una imatge difícil de percebre). Però això es podria solucionar utilitzant la funció *glPointSize*, que bàsicament modifica la mida d'una primitiva del tipus *GL_POINTS*, amb un valor que es buscaria empíricament, de manera que aconseguís que la imatge es pogués veure en condicions acceptables.

Problema

Tot i ser únicament una idea inicial i no havia de ser per res el mètode definitiu, aquí es va presentar un obstacle important: **en OpenGL ES no es pot definir geometria utilitzant els mètodes *glBegin* i *glEnd***.

Això implica que a l'hora de cridar la funció de renderitzat, tota geometria ha d'haver estat definida abans ja. A part de simplement invalidar el mètode pensat, això també va provocar arribar a una conclusió: si s'havia de tenir la geometria ja definida (d'alguna manera, de moment es desconeixia com), això volia dir més memòria necessària. Havent eludit ja un *memory warning*, no era una conclusió massa agradable.

[18] Per pintar punts (cada vèrtex definit seria un punt). Altres possibles valors són *GL_LINES* (per pintar línies definides per 2 vèrtexs), *GL_QUADS* (per pintar polígons de 4 vèrtexs), *GL_TRIANGLES* (per pintar triangles definits amb 3 vèrtexs), etc. Més informació a http://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml

3.2.3.2 Filtre ITK

Davant la conclusió extreta en l'anterior apartat, una de les següents idees que va passar pel cap va ser el d'intentar crear un nou filtre ITK per tal d'aconseguir transformar la imatge ITK ja en memòria en un format que pogués ser representable més fàcilment per OpenGL.

Per definir un filtre s'havien de definir, entre moltes altres coses, el tipus d'imatge d'entrada, el tipus d'imatge de sortida i la funció principal que transformés la imatge d'entrada en el format de la de sortida.

El nou filtre podria servir per dues coses totalment diferents:

- En un punt inicial, es pretenia transformar la informació de luminància de la imatge en informació RGBA.
- Després es va pensar que per cada píxel, es podia mirar de crear un punt amb 3 coordenades, que a més contingués la informació de color (una tupla amb coordenades de l'espai i color).

La primera idea era, a partir de la imatge llegida en format luminància, generar un seguit de píxels en format RGBA[19]. Caldria reduir el nombre de píxels total, ja que si es mantenia la mida de la imatge original, creixeria la memòria utilitzada a nivells inviables:

Si per cada *unsigned char* es passés a tenir 4 *floats* (un per component d'RGBA), la memòria que caldria seria 4 vegades superior.

Per reduir la imatge es podria haver utilitzat els iteradors per interpolar manualment el valors d'*n* píxels, agafant-ne un com a centre i utilitzant els del seu voltant per interpolar, de manera que es reduís la mida total de la imatge. Ara bé, això hagués suposat una pèrdua en la qualitat de la imatge. No se sabria del cert si seria una pèrdua de qualitat excessiva fins que no es posés a prova.

El problema era que això no resolia el problema de la representació. Encara caldria definir una geometria a la que aplicar aquesta informació de color.

L'altra opció, la de transformar cada píxel de la imatge en un punt geomètric obria dos problemes: com definir a l'espai el punt (com distanciar els punts) i a més, el típic problema de memòria: si per cada píxel del que es tenia un sol valor de luminància s'afegien 3 coordenades a l'espai (que forçosament havien de ser *float*), la memòria que ocuparia la imatge en global seria excessiva. Ni tant sols amb la interpolació que es podria fer (la proposada en la primera idea) hagués servit, ja que per cada píxel s'estarien afegint 4 bytes (mida float) * 3 floats = 12 bytes de més per píxel... impensable.

Problemes

Encara que d'alguna manera es poguessin solucionar els problemes presentats anteriorment, la complexitat d'afegir un filtre va ser un problema més. Costava

[19] RGBA: Red Green Blue Alpha: Intensitat de colors vermell, verd i blau, respectivament, i opacitat.

d'entendre com s'havia d'afegir un nou filtre i la documentació d'ITK no era prou entenedora. S'hi va dedicar un temps però es va descartar relativament ràpid.

De fet, després d'analitzar més a fons com funcionaria el sistema, es va veure que no seria viable fins i tot si es solucionaven els problemes de memòria, ja que un cop transformada la imatge llegida al format necessari per a la representació amb el nou filtre, s'obrien dues opcions:

- Es mirava de definir els filtres ITK que es preveien utilitzar de manera que poguessin operar amb l'estructura definida pel nou filtre (no era segur que d'entrada poguessin).
- S'havia de fer el filtre invers, és a dir, transformar la imatge en format representable al format anterior, en el resultat de llegir la imatge amb el lector.

De no fer cap de les dues coses, encara que es pogués representar la imatge, no se li podria aplicar cap filtre, ja que un cop transformada en l'estructura representable es perdria la imatge original (calia optimitzar en memòria i no es podia mantenir dues imatges simultànies a memòria). Per tant, es va descartar com a mètode viable.

De fet, es va utilitzar l'eina Instruments altre cop, per assegurar-se que quan s'aplica un filtre ITK, la imatge no es modifica, sinó que se'n crea una de nova:

Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes
☑	* All Allocations *	19,79 MB	20983	0	19,79 MB

Figura 3.2.3.2.1: Sortida de l'eina instruments al aplicar un filtre a la imatge llegida

Efectivament, al aplicar un filtre es dobla la memòria usada. De fet, al fer aquest anàlisi l'aplicació es tanca. Això pot voler dir que excedeix la memòria que l'iOS li cedeix.

Això és una mala notícia. Implicava que, independentment de com acabéssim representant en 3D la imatge ITK, si li volíem aplicar un filtre ocuparíem el doble de memòria. Si no s'hagués fet la conversió del tipus de la imatge de *short* a *unsigned char*, hagués estat inviable.

3.2.3.3 ITK Mesh

Un altre mesura que es va plantejar va ser la d'utilitzar les *Mesh* (malles) d'ITK. Les malles estan pensades per representar formes en l'espai. Deriven de la classe *itk::PointSet*, per tant tenen totes les funcionalitats relacionades amb punts i accés a informació per píxel relacionada amb els punts. La *mesh* pot ser de dimensió *n*, amb el que ofereix molta flexibilitat.

Les malles poden estar formades per cèl·lules o *cells*, i cada una de les cèl·lules pot ser del tipus *itk::LineCell*, *itk::TriangleCell*, *itk::QuadrilateralCell* o *itk::TetrahedronCell*.

Així doncs, el que es volia fer era transformar la imatge ITK a una malla, que podria ser compatible amb la geometria d'OpenGL, i llavors representar-la. Per fer això caldria primer mirar si existia la possibilitat de fer una transformació automàtica, potser a través d'un filtre dels que proporciona la pròpia llibreria.
De no ser possible, seria molt més complex: caldria definir una estructura geomètrica a partir dels punts dels quals només es tenia la intensitat de lluminositat. Per a fer-ho, s'hagués intentat crear cèl·lules del tipus *TriangleCell* (triangular), de manera que es dividís la imatge en triangles, amb la informació geomètrica dels tres punts que la formessin i del color d'aquests tres punts interpolat (per evitar tenir un color per punt, per estalviar memòria).

Aquest procés hagués sigut feixuc.

Problema

No es va aconseguir pel mateix motiu que abans, no acabava d'estar clar com funcionava i no era fàcil d'entendre. Es va buscar documentació de com fer aquesta transformació però el màxim que es trobava eren preguntes a fòrums sense respostes clares.

I per si fos poc, la llibreria ITK donava problemes depenent de quin fitxer de capçaleres s'inclogués. A vegades el sol fet de compilar i llençar l'aplicació ja suposava un gran esforç: no se sabia si era que faltava incloure fitxers o era que la llibreria inclosa no era al 100% estable.

A més, tot i que s'hagués aconseguit, tenia el mateix problema que el filtre anterior:

Per defecte no era una transformació bidireccional, amb el que un cop transformada la imatge en malla, seria difícil aplicar-li filtres i després imprimir-la altre cop. A més, es va arribar a la conclusió de que utilitzant les *cells* per representar els punts, hagués augmentat de manera inviable la memòria necessària per mantenir la imatge. Mesura descartada.

3.2.3.4 Shaders

Arribats a aquest punt, semblava que no es podia comptar amb ITK, així que la responsabilitat de representar la imatge havia de recaure únicament en OpenGL i les seves funcionalitats. Així doncs, havia arribat el moment de pensar com podia OpenGL permetre renderitzar la imatge que ja teníem en memòria però que d'allà no aconseguíem representar.

La solució semblava utilitzar *shaders*. El següent apartat descriu com es van investigar per a la seva utilització.

3.2.4 Anàlisi de la viabilitat d'ús de shaders

3.2.4.1 Què és un shader?

Un *shader* és un conjunt d'instruccions dissenyat per executar-se en el processador gràfic, en algun dels punts programables de la *rendering pipeline* (procés de renderitzat).

Molts cops és simplement un *string*, que posteriorment serà compilat i afegit a un objecte *program*. Més detalls de com es van afegir *shaders* en el codi actual d'iOS en la secció 3.2.5.8.

3.2.4.2 Fixed pipeline vs Programmable pipeline

El procés de renderitzat tradicional és l'anomenat *fixed pipeline*. En la següent imatge es pot veure el procés que es segueix per renderitzar una determinada escena definida en una aplicació qualsevol.

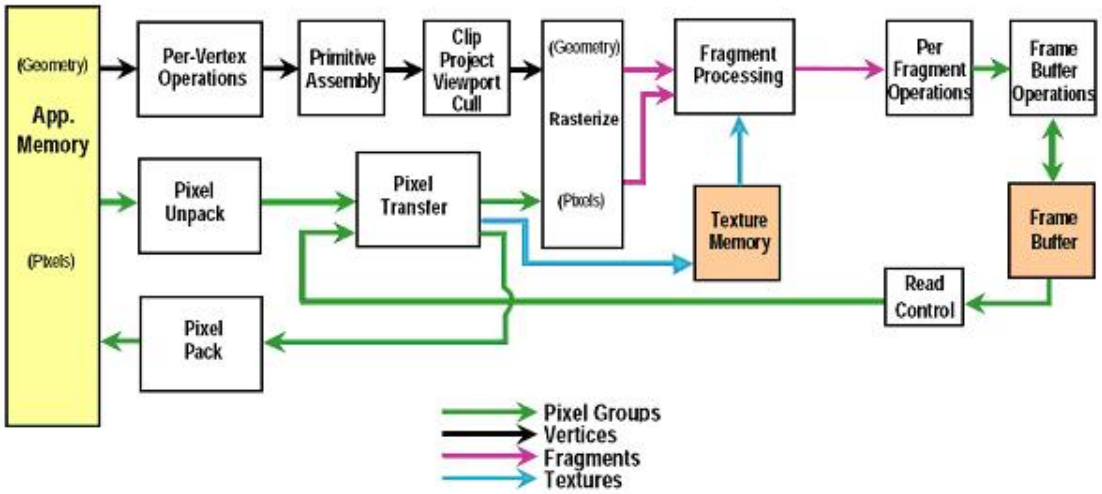


Figura 3.2.4.2.1: Fixed pipeline d'OpenGL

No seria adequat ni necessari ara descriure en profunditat el procés de renderitzat d'OpenGL, només ressaltar uns quants punts:

Utilitzant les funcions definides a l'API d'OpenGL tals com, per exemple, *gluLookAt* o *glLightfv,* el que es fa, al renderitzar, és passar per aquest *pipeline*, delegant la responsabilitat de traduir el codi de les funcions en codi apte pel processador gràfic al compilador.

Aquestes funcions permeten a l'usuari aplicar models senzills d'il·luminació i efectes també senzills, com alguns reflexos.

Això sí, per a fer-ho, cal utilitzar tècniques com el *multi-texturing* (utilitzar més d'una textura[20] simultàniament) i/o múltiples passades (un mateix polígon ha de passar per tot el procés de renderitzat més d'un cop).

[20] Una textura es pot imaginar com una imatge que es superposa a un polígon per donar-li un nivell més de de detall.

Com a punt a favor, aquestes funcions són relativament senzilles d'utilitzar. Els seus paràmetres tenen uns possibles valors força limitats i aquests estan ben documentats.

Ara bé, tot i que poden ser suficients per l'aplicació que es pot tenir previst desenvolupar, limiten bastant les accions possibles.

Per tant, va sorgir la necessitat de proporcionar més llibertat als programadors, i això es va aconseguir amb la *programmable pipeline*. La *programmable pipeline* és semblant a la *fixed pipeline* però es diferencia perquè algunes parts del procés són substituïdes per *shaders* escrits pel propi programador.

Un *shader*, amb poques instruccions, és capaç de fer el mateix que fa la funció corresponent del *fixed pipeline*. De fet, ho ha de fer: si s'utilitza un *shader* i no es volen perdre les funcionalitats bàsiques, les que faria el *fixed pipeline*, cal que el *shader* les implementi, d'altra banda per sí soles no es faran, ja que un *shader* d'un determinat tipus substitueix completament a la part del procés corresponent en el *fixed pipeline*.

Però això en realitat també ofereix una possibilitat: la d'escollir si algun dels càlculs o transformacions que es fan per defecte amb el *fixed pipeline* es vol fer o no. A més, també dóna opció a fer més transformacions i càlculs que el programador vegi oportuns per aconseguir efectes molt més espectaculars dels aconseguits amb les instruccions bàsiques de l'API.

Un altre punt a favor dels *shaders* és que s'executen a la GPU, i aquesta alhora pot treballar en paral·lel per diferents unitats independents (més d'un vèrtex alhora, per exemple, en el cas d'un *vertex shader*) gràcies al hardware de la GPU predisposat al paral·lelisme.

Això fa que, tot allò que pugui ser traduït de codi que s'executaria a la CPU a codi que s'executarà a la GPU (*shader*), sigui molt més eficient executant-se a la GPU.

En la següent secció s'expliquen breument els shaders més rellevants que existeixen.

3.2.4.3 Tipus de shaders

Vertex shaders

Un *vertex shader* s'executa per cada vèrtex de la geometria definida. Quan s'activa l'ús d'un *vertex shader*, aquest s'executarà en lloc de les operacions pre-vèrtex prefixades d'OpenGL, per tant, en la majoria dels casos, caldrà que el *shader* les implementi.

En la figura 3.2.4.2.1, el *shader* substituirà la caixa "Per-Vertex Operations".

Alguns dels atributs interessants que arriben al *shader* són:

- gl_Vertex: conté les coordenades del vèrtex en el sistema de referència del model.
- gl_Normal: conté la normal del vèrtex.
- gl_MultiTexCoord{0-n}: conté les coordenades de la textura corresponent.
- Variables definides per l'usuari.

I què s'espera d'un *vertex shader*? Doncs que, com a mínim:

- Doni valor a la variable gl_Position, que conté les coordenades del vèrtex en coordenades de *clipping*[21] .
- Doni valor a les variables d'usuari que s'hauran de propagar al següent punt del procés, a un altre *shader* definit per un usuari.

A més, també es pot transformar i normalitzar la normal del vèrtex, en *eye space*[22], calcular la il·luminació del vèrtex i generar les coordenades de textura del vèrtex.

Tots els valors de variables que arribin al següent punt *shader* (en aquest cas pot ser un *geometry shader* o un *fragment shader*) arriben per interpolació.

Geometry shaders

Aquest tipus de *shader* és relativament nou.

S'executa just després dels *vertex shader*.

Com a input rep una primitiva sencera. Llavors, a partir d'elles, el *geometry shader* pot emetre una o més primitives, que seran rasteritzades[23] i passades al *fragment shader*.

Les noves primitives només poden ser punts, línies o tires de triangles (triangles que comparteixen una cara dos a dos) a partir d'aquelles que han sigut enviades a l'inici del procés de renderitzat.

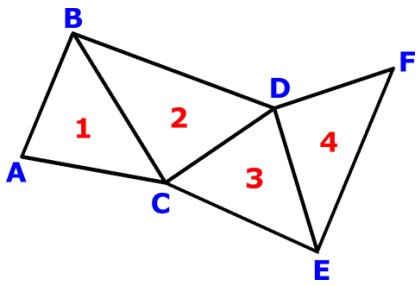


Figura 3.2.4.3.1: Tira de triangles

També pot modificar primitives, separant-les o eliminant vèrtexs.

---

[21] Sistema de coordenades que decideix si un vèrtex queda dins o no de la part de l'escena a representar.
[22] Sistema de coordenades on l'origen és la càmera de l'escena.
[23] Rasterització és el procés pel qual cada primitiva individual és dividida entre elements discrets, fragments, per tal de ser processats individualment per determinar-ne el color final.

Fragment shaders

Un *fragment shader* s'executa per cada fragment a pintar. Un fragment és una porció d'una primitiva rasteritzada.

Té com a objectiu calcular el color final d'un fragment i un valor de profunditat per aquest.

Quan s'activa l'ús d'un *fragment shader*, aquest s'executarà en lloc de les operacions per-fragment prefixades d'OpenGL, en la majoria dels casos, caldrà que el *shader* les implementi.

En la figura 3.2.4.2.1, el *shader* substituirà la caixa "Fragment Processing".

Alguns dels atributs interessants que arriben al *shader* són:

- gl_TexCoord{0-n}: conté les coordenades de textura corresponent interpolades.
- gl_FragCoord: conté les coordenades del fragment en coordenades de dispositiu[24].
- Variables definides per l'usuari.

I què s'espera d'un *fragment shader*? Doncs que, com a mínim:

- Doni valor a la variable gl_FragColor, que conté el color final amb el que es representarà el fragment.

A més també pot utilitzar les textures per aplicar-les al fragment, o incorporar efectes a nivell de fragment, per exemple boira.

Ara bé, un *fragment shader* **no** pot canviar les coordenades del fragment, ni accedir a la informació d'altres fragments.

[24] Les coordenades de dispositiu són aquelles que només tenen dos valors x i y, que corresponen a la ubicació d'un determinat fragment en la finestra de visió (equivalent a la pantalla del dispositiu, si la finestra l'ocupa tota).

3.2.4.4 GLSL

El GLSL (*OpenGL Shading Language*) és el principal llenguatge d'alt nivell utilitzat per programar *shaders*. Tot i que més llenguatges disponibles, és l'únic que forma part del nucli d'OpenGL.

Té un estil semblant al C/C++ en bastants aspectes, degut a que la sintaxi està basada en el C. Tot i així també hi ha diferències notòries:

- No hi ha conversió automàtica de tipus.
- No suporta apuntadors, el tipus *char*, el tipus *double*, el tipus *short* ni el tipus *long*.
- Les funcions tenen paràmetres d'entrada *in*, paràmetres de sortida *out* i paràmetres d'entrada/sortida *inout*. En tots els casos es passen per valor.

Va ser creat per OpenGL ARB[25] per donar més llibertat i control sobre el procés de renderitzat als desenvolupadors, sense haver de fer servir el llenguatge assemblador d'ARB o llenguatges específics del hardware utilitzat.

Va ser introduït com a extensió en la versió OpenGL 1.4 i va ser inclòs al nucli d'OpenGL en la versió 2.0.
Alguns dels beneficis d'utilitzar GLSL són:

- Compatibilitat multiplataforma en múltiples sistemes operatius, Windows, Linux i Mac OS X inclosos.
- La possibilitat d'escriure *shaders* que poden ser utilitzats en qualsevol tarja gràfica que suporti el GLSL (és a dir, un mateix *shader* serveix per totes les diferents targes que suportin el llenguatge).
- Cada venedor inclou el compilador de GLSL en el seu driver, per tant, permet que el propi venedor optimitzi el codi corresponent segons l'arquitectura de la seva tarja gràfica.

Suporta les estructures de flux habituals, els condicionals *if*, els bucles *for*, *while*, però alhora té diferències en el llenguatge, com per exemple la instrucció *discard*, que s'utilitza per descartar el fragment.

Per més detalls sobre el llenguatge GLSL, consultar l'annex 1.

[25] OpenGL Architecture Review Board: és un consorci industrial que regeix l'especificació d'OpenGL. Al 31 de juliol del 2006 va transferir el control de l'especificació d'OpenGL al grup Khronos.

3.2.5 Inclusió de shaders en el projecte

3.2.5.1 Limitacions OpenGL ES 2.0

Incloure *shaders* en l'aplicació semblava que podria ser la solució a la necessitat que hi havia de mostrar la imatge en tres dimensions, donada la llibertat que proporcionen.

De fet, un dels usos més comuns d'un *geometry shader* és el de crear una "malla de filferro" (*wire-frame*) en una sola passada de renderitzat. Podria servir per formar nova geometria en la GPU, aconseguint així que només ocupés memòria d'aquesta, i evitant per tant el *memory warning* que podria provocar el tenir geometria complexa a la CPU.

Al investigar una mica, però, es va descobrir que els *geometry shaders* no són compatibles amb OpenGL ES 2.0.

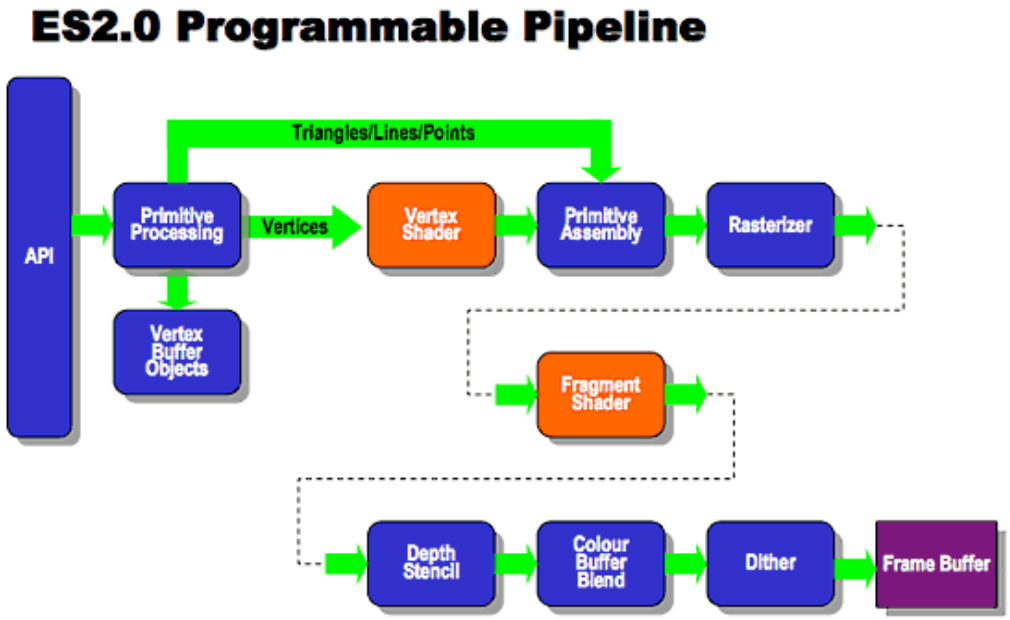


Figura 3.2.5.1.1: Pipeline programable d'OpenGL ES 2.0

Com es pot observar a l'anterior figura, només són possibles els *vertex* i *fragment shaders*, així que hauran de ser suficient pel propòsit que ens ocupa.

A més, una dificultat addicional afegida per OpenGL ES, és que els *shaders* no disposen de la majoria de variables predefinides útils de les que es disposa en OpenGL normal. Variables com poden ser, per exemple, la matriu de visualització.

3.2.5.2 Com aplicar shaders

Un cop havent donat un cop d'ull al què ofereix el *programmable pipeline*, i més concretament a la d'OpenGL ES 2.0, i als *shaders* i a la seva sintaxi (a l'annex 1), com es podia aplicar tot l'après al problema que es presentava, de representar en tres dimensions la imatge ITK llegida i en memòria?

Per respondre a aquesta pregunta primer calia respondre'n unes altres:

Com es pot fer arribar la informació de la imatge ITK al shader? Cal enviar-la tota? Es pot, d'alguna manera, enviar només una part necessària?

Però abans fins i tot de ser capaç de respondre a aquestes qüestions, el primer que calia era incloure els *shaders* al projecte.

3.2.5.3 Diferències en iOS

Tot i que el GLSL és un llenguatge estàndard, la pròpia pàgina de desenvolupadors d'Apple presenta algunes pautes necessàries a dur a terme per tal de poder utilitzar *shaders*. Per exemple, obliga a utilitzar prefixos de precisió en la declaració de les variables. On a qualsevol lloc es podria escriure quelcom com:

```
vec4 v = vec4(0.0);
```

S'obliga a escriure:

```
highp vec4 v = vec4(0.0);
```

O equivalents amb altres prefixos de precisió.

La majoria d'aquests canvis són més aviat orientatius per tal de millorar l'eficiència i afectaven poc a la manera de com s'havien d'escriure els *shaders*, però calia tenir-los en compte.

3.2.5.4 Geometria necessària

Donat que no es podia crear geometria extremadament complexa, pel tema del *memory warning*, i tampoc es podia utilitzar cap filtre o funcionalitat d'ITK que facilités la representació de la imatge, el que calia era definir una geometria senzilla i llavors, a través de l'ús de *shaders*, aconseguir que, d'alguna manera, es ressaltés o es modifiqués la geometria per tal de veure el que interessa, la imatge en 3D.

Però recordem que el problema que ens havíem trobat era que les instruccions d'OpenGL per definir geometria, *glBegin* i *glEnd*, no eren vàlides en OpenGL ES.

Així doncs, com passar la geometria al *shader*? Utilitzant les anomenades *vertex arrays* conjuntament amb els anomenats *vertex buffer objects*.

3.2.5.5 Vertex Arrays

Les arrays de vèrtex o *vertex arrays objects* (VAO) el que fan és contenir tot l'estat necessari per especificar els vèrtexs de la geometria a l'hora de renderitzar. Defineixen el format del vèrtex així com la font de la informació de vèrtexs (ells no contenen les dades dels vèrtexs per sí sols, tenen referències cap a elles a través de punters).

Com qualsevol altre objecte d'OpenGL, els VAO disposen de les funcions de creació, destrucció i vinculació (*bind*) usuals:

- glGenVertexArrays com a funció creadora.
- glDeleteVertexArrays com a funció destructora.
- glBindVertexArray com a funció vinculant.

Cada VAO té un nombre d'atributs, que estan indexats de 0 al valor de la constant GL_MAX_VERTEX_ATTRIBS (valor que depèn del hardware).

Per defecte, cada atribut està deshabilitat, cal habilitar els atributs amb la funció glEnableVertexAttribArray. Com sembla lògic, també es poden deshabilitar de nou els atributs amb la funció glDisableVertexAttribArray.

A cada atribut se li pot assignar la informació corresponent a algun dels atributs d'un vèrtex (posició, normal, coordenades de textura, etc.) utilitzant la funció glVertexAttribPointer. No hi ha cap mena d'ordre estipulat, cosa que vol dir que, mentre l'índex utilitzat a glVertexAttribPointer hagi estat activat abans amb glEnableVertexAttribArray (és necessari que s'hagi activat), tant fa si aquest índex és el 0 o l'1 o qualsevol altre.

Cal dir que normalment s'utilitzen *vertex buffer objects* per passar les dades de vèrtexs amb glVertexAttribPointer, però això no té perquè ser així, també es poden utilitzar arrays comunes definides per l'usuari.

3.2.5.6 Vertex Buffer Objects

Abans de definir què és un *vertex buffer object* (VBO), primer definim què és un *buffer object*:

Un *buffer object* és un objecte proporcionat per OpenGL que emmagatzema informació en forma d'array i que la memòria que utilitza és reservada en el *context*[26] d'OpenGL (és a dir, a la GPU).

Un *buffer object* pot contenir des d'informació de vèrtexs, a informació de píxels extrets d'imatges (o del *framebuffer*[27]) i algunes coses més[28].

Així doncs, un *vertex buffer object* és un dels tipus de *buffer object*, el que s'utilitza per emmagatzemar informació de vèrtexs (de primitives).

---

[26] L'objecte *context* d'OpenGL representa vàries coses. Conté tot l'estat de la instància d'OpenGL actual. Bàsicament, si l'objecte *context* és destruït, OpenGL és destruït.

[27] Un *framebuffer* és una col·lecció de *buffers* que poden ser utilitzats com a destinació al renderitzar. OpenGL té dos tipus de *framebuffers*: el per defecte, que ve donat al crear un *context*, i els creats per l'usuari a través dels anomenats *framebuffer objects*. Més informació a la referència extra [10].

[28] Més informació a la referència extra [11].

3.2.5.7 Aplicació de VAO més VBO

Un cop analitzats, al utilitzar VBO per emmagatzemar la informació de la geometria i VAO per mantenir l'estat d'OpenGL, vam ser capaços de definir geometria.

Però... quina geometria? Havia de ser un polígon que pogués contenir qualsevol tipus d'imatge mèdica en el seu interior i que sigui el més simple possible, ja que la part de perfilar la imatge ITK correrà a càrrec dels *shaders*. La solució més òbvia era doncs dibuixar un cub que servís com a caixa contenidora.

Per a crear un cub, doncs, primer vam definir la informació dels vèrtexs:

Havent creat l'estructura:

```
typedef struct {
    float Position[3]; //Coordenades x, y i z del vèrtex
    float TextureCoordinates[3]; //Coordenades de textura
}Vertex;
```

Crear els vèrtexs correspon a fer:

```
Vertex v1 = {{-1.5, 1.5, 1.5}, {0.0, 1.0, 1.0}};
Vertex v2 = {{-1.5, -1.5, 1.5}, {0.0, 0.0, 1.0}};
Vertex v3 = {{1.5, 1.5, 1.5}, {1.0, 1.0, 1.0}};
Vertex v4 = {{1.5, -1.5, 1.5}, {1.0, 0.0, 1.0}};

Vertex v5 = {{-1.5, 1.5, -1.5}, {0.0, 1.0, 0.0}};
Vertex v6 = {{-1.5, -1.5, -1.5}, {0.0, 0.0, 0.0}};
Vertex v7 = {{1.5, 1.5, -1.5}, {1.0, 1.0, 0.0}};
Vertex v8 = {{1.5, -1.5, -1.5}, {1.0, 0.0, 0.0}};
```

I definir el cub és simplement definir les seves cares:

```
Vertices = malloc(sizeof(Vertex) * numberVertices);

//Cara frontal
Vertices[0] = v1;
Vertices[1] = v2;
Vertices[2] = v3;
Vertices[3] = v2;
Vertices[4] = v4;
Vertices[5] = v3;

//Cara de darrera
Vertices[6] = v6;
Vertices[7] = v5;
Vertices[8] = v7;
Vertices[9] = v7;
Vertices[10] = v8;
Vertices[11] = v6;

//Cara esquerra
Vertices[12] = v1;
Vertices[13] = v5;
```

```
Vertices[14] = v6;
Vertices[15] = v6;
Vertices[16] = v2;
Vertices[17] = v1;

//Cara dreta
Vertices[18] = v3;
Vertices[19] = v4;
Vertices[20] = v8;
Vertices[21] = v8;
Vertices[22] = v7;
Vertices[23] = v3;

//Cara superior
Vertices[24] = v1;
Vertices[25] = v3;
Vertices[26] = v7;
Vertices[27] = v7;
Vertices[28] = v5;
Vertices[29] = v1;

//Cara inferior
Vertices[30] = v2;
Vertices[31] = v6;
Vertices[32] = v8;
Vertices[33] = v8;
Vertices[34] = v4;
Vertices[35] = v2;
```

Això el que farà és definir un cub com el següent:

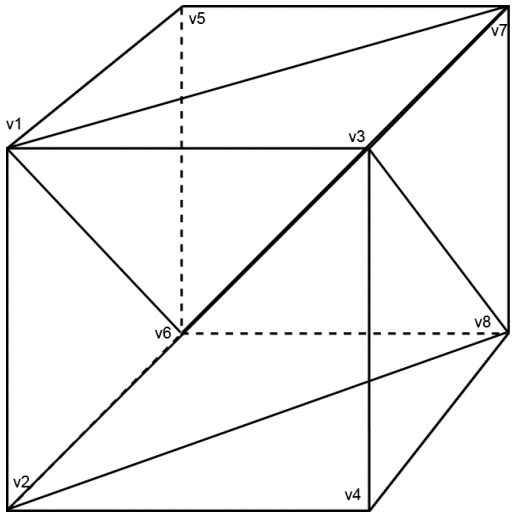


Figura 3.2.5.7.1: El cub que hem definit. Cada cara està dividida en dos triangles.

De costat 3, ja que les coordenades van de -1.5 a 1.5.

I on les coordenades de textura segueixen la filosofia típica:

- La cara esquerra del cub té coordenada de textura 0 per l'eix X, mentre que la cara dreta té coordenada 1 pel mateix eix.
- La cara inferior del cub té coordenada de textura 0 per l'eix Y, mentre que la cara superior té coordenada 1 pel mateix eix.
- La cara posterior té coordenada de textura 0 per l'eix Z, mentre que la cara frontal té coordenada 1 pel mateix eix.

Pot resultar curiós el fet d'haver fet servir triangles per dividir les cares del cub, però això és així perquè, un cop més, OpenGL ES té alguna cosa a dir: la funció que es fa servir per donar l'ordre de renderitzar la geometria, *glDrawArrays*, no accepta com a tipus de primitiva els quadrilàters, només es poden pintar punts, línies, tires de triangles o ventalls de triangles (triangles que comparteixen una cara dos a dos i tots tenen un vèrtex en comú).

Després s'apliquen crides necessàries a *glEnableVertexAttribArray* i *glVertexAttribPointer*, a les quals no cal entrar en detall, de tal manera que el cub queda definit.

Com es pot comprovar, el grau de complexitat de només definir la geometria és força elevat, si es compara amb la senzillesa que suposava crear polígons simplement amb *glBegin*, unes quantes crides de *glVertex3f* per definir vèrtexs i *glEnd* en acabat. Per sort, havent de definir una geometria tant senzilla, no ens afecta massa negativament.

Un inconvenient d'acabar utilitzant aquest sistema, però, és que no dóna la opció d'optimitzar l'espai al driver, ja que les dades (vèrtexs) estan referenciades mitjançant punters al VBO i poden variar.

3.2.5.8 Unió dels shaders amb el projecte

Per definir un *shader*, el que es fa és escriure'l en un fitxer a part i és considerat un string més (com si el fitxer només contingués una cadena de caràcters).

Llavors, a través d'un objecte *program*, es relacionen els diferents tipus de *shader*, de manera que en temps d'execució es compilen i linken a aquest objecte.

Aquest objecte pot tenir un o varis *shaders* associats i té el seu propi log d'errors en cas de que alguna cosa hagués fallat durant la compilació (o linkatge).

Per incloure els *shaders* es van seguir els passos especificats a l'enllaç de la pàgina de desenvolupadors d'Apple corresponent (inclòs a la referència extra [12]), que van permetre utilitzar els *shaders* en el projecte en l'Xcode que teníem aleshores.

3.2.5.9 Debugar en shaders

Debugar en *shaders* no és tasca fàcil. No hi ha consola per on mostrar errors, no hi ha *breakpoints*, no hi ha pràcticament res. Així doncs, com es debuga? Doncs amb el que fa un shader: pintar.

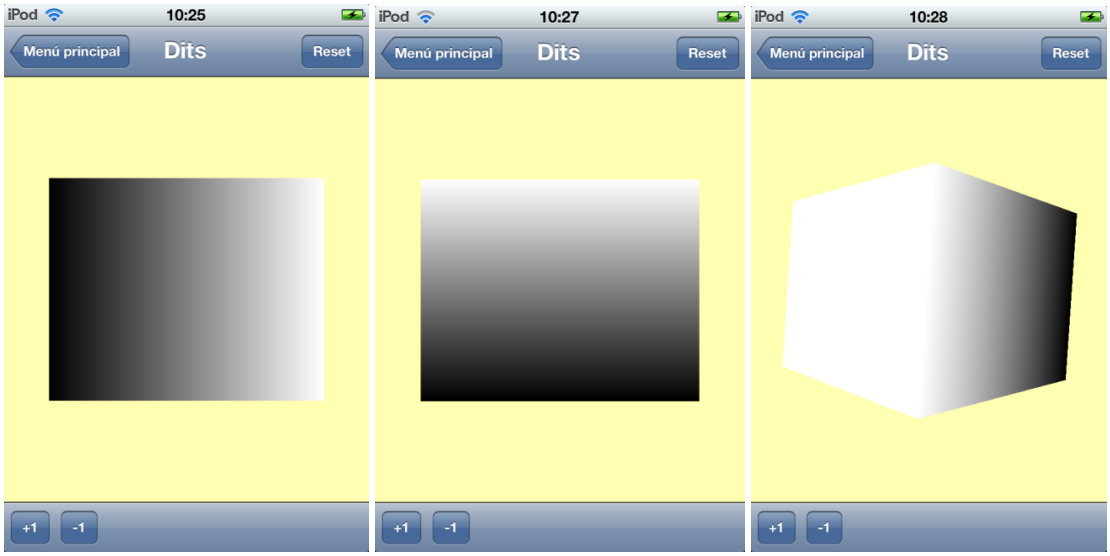
El que es fa per debugar, doncs, és renderitzar utilitzant patrons que puguin ser reconeguts per tal de saber què està fent exactament el codi GLSL.

Per exemple, en un punt determinat del projecte, va caldre comprovar que les coordenades de textura arribaven amb el valor esperat al *fragment shader*. Doncs per fer-ho, podem utilitzar un *fragment shader* tal que el *main* té la forma:

```
void main()
{
      gl_FragColor = vec4(vec3(textureVarying.coordinate), 1.0);
}
```

On *textureVarying* és la variable amb la que arriben els valors de les coordenades de textura, i *coordinate* ha de valer x, y o z, en funció de quina coordenada vulguem comprovar.

Llavors, al executar l'aplicació i observar els resultats:



Figures 3.2.5.9.1, 3.2.5.9.2 i 3.2.5.9.3: Representacions visuals de les coordenades de textura segons l'eix X, segons l'eix Y i segons l'eix Z respectivament.

Es pot observar l'escala de grisos segons l'eix corresponent. Tenint en compte que el color determinat pel vector (1.0, 1.0, 1.0, 1.0) és blanc, i el color (0.0, 0.0, 0.0, 1.0) és el negre, té sentit que la coordenada de textura X evolucioni de tal manera que a l'esquerra els píxels estiguin de color negre (on aquesta val 0) i a la dreta estiguin blancs (on val 1). El mateix passa per les coordenades de textura corresponents als eixos Y i Z.

Aquest és un exemple de com cal debugar un *shader*. Si bé aquest exemple era molt senzill, i la manera de fer la comprovació és immediata, quan el codi del *shader* es complica no resulta tant senzill imaginar una possible renderització que ajudi a veure què pot estar fallant en el codi.

A més, una limitació important és el fet de que els valors que pot prendre una component d'un color han d'estar compresos entre 0.0 i 1.0, per tant si es vol debugar el valor d'una variable amb un rang de valors diferent cal fer les modificacions necessàries per tal de adaptar el rang de la variable a [0.0, 1.0].

Amb tot, debugar un *shader* no es cosa fàcil i pot resultar una tasca feixuga, d'aquí que una de les coses que ha consumit més temps del projecte ha estat la confecció del *shader* responsable de la representació en tres dimensions de la imatge ITK.

3.2.6 Utilització de Linux amb màquina virtual

Abans de començar amb les següents seccions, és important remarcar que durant el procés de desenvolupament del projecte van provar-se diverses coses, i com que el coneixement que es tenia de *shaders* i d'OpenGL en general era força limitat, de seguida van sorgir problemes alhora de fer servir el que s'explicarà en les següents seccions.

El problema més greu va ser que l'iPod, dispositiu on fins llavors s'havia corregut sempre l'aplicació, deixava de ser capaç de reproduir res.

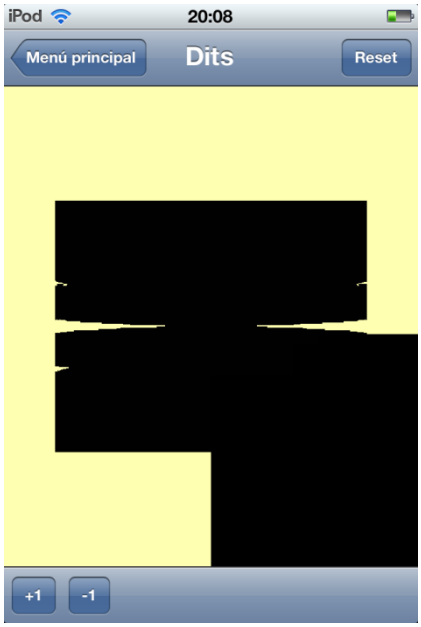


Figura 3.2.6.1: Captura de pantalla de l'iPod

Per aconseguir la captura de la figura anterior va caldre rapidesa de dits, ja que depenent de quina execució, l'aplicació es tornava inestable i es comportava de manera pseudoaleatòria. El poder haver aconseguit una captura prou decent és pura sort.

La mesura que es va prendre llavors va ser utilitzar una màquina virtual amb Ubuntu instal·lat en ella. Amb ella, i juntament amb un codi visualitzador del que ja es disposava (implementava una finestra amb OpenGL i poca cosa més), es van anar fent les proves deixant de banda l'iPod, que no era capaç de córrer un mateix *shader* perquè simplement es bloquejava: no podia arribar a fer ni un frame per segon.

Això ha provocat que moltes de les conclusions a les que s'ha arribat en les següents seccions s'hagin hagut d'extraure primer fent servir aquesta màquina virtual amb Ubuntu i mirant després d'extreure-ho a la versió d'iOS.

3.2.7 Textures

Un cop els *shaders* van ser inclosos, el que es va pensar va ser com passar la imatge ITK a aquests. La resposta va venir ràpid: amb textures.

3.2.7.1 Definició de textura

Una textura és un objecte d'OpenGL que conté una o més imatges. Una textura pot servir per dues coses: o per ser accedida des d'un *shader* o pot ser una destinació de renderitzat (en comptes de pintar per la pantalla del dispositiu, s'aboca la informació de color en una textura).

Així doncs, es podrien fer servir textures per passar la imatge ITK als *shaders*.

Ara bé, sorgia un petit problema: la imatge original tenia 256x256x139 elements, però deguda la necessitat d'utilitzar textures, la majoria exigien que les dimensions fossin potència de 2, així que per simplificar, en molts dels casos exposats a continuació, es va assumir que la imatge en realitat tenia 256x256x128 elements.

Com es veurà més endavant, aquesta simplificació serà molt útil a l'hora de fer arribar la imatge al *shader*.

3.2.7.2 3D textures

La primera opció va ser utilitzar textures 3D. Aquestes textures, com diu el nom, són de 3 dimensions i servirien perfectament pel propòsit que ens ocupa: la imatge ITK que tenim en memòria pot té exactament la mateixa estructura que una textura 3D d'OpenGL. Seria ideal poder-la passar al *shader* utilitzant aquest tipus d'objecte, doncs.

La idea seria enviar a pintar el cub que hem definit abans tal qual, sense res més que els vèrtexs. Llavors, en un *fragment shader,* utilitzar la funció de GLSL *texture3D*, i les coordenades de textures assignades als vèrtexs, per poder accedir fàcilment a la textura i, per cada fragment, pintar el color corresponent.

Això, però, no acabaria de funcionar si ho féssim només d'aquesta manera. Cada tall de la imatge ITK té una part negre que equival a transparència (com més proper al color negre, més transparent representa aquell punt del tall corresponent). I això que implica? Doncs que el *fragment shader* s'executa un cop per fragment (~píxel) a pintar. Si en aquell punt, la imatge ITK tingués un punt negre, o sigui, transparència, seria incorrecte descartar directament el fragment, ja que, seguint la direcció de visió, podria ser que ens trobéssim un altre tall de la imatge que no fos transparent.

Així doncs, si s'utilitzés simplement una textura 3D i el cub, el que acabaria passant és que es veurien els punts no transparents externs només. Si d'alguna manera es definís que els punts negres no es pintessin, fossin transparents, llavors el que passaria és que en aquell punt no hi hauria color, però no es comprovaria el mateix punt pel següent tall.

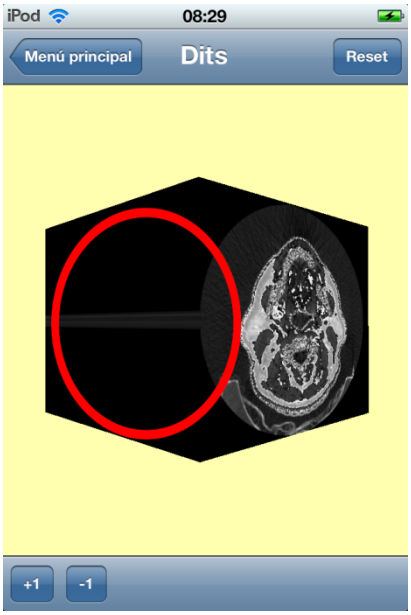


Figura 3.2.7.2: Si es descartessin els fragments encerclats, perquè aparentment la imatge és negre (transparent), s'estaria descartant erròniament informació visual.

Però llavors, de què serviria tenir una textura 3D? Sobre les estratègies de com utilitzar la imatge, un cop passada al *shader* se'n parlarà a la secció 3.2.9.

Problema

Tot i ser la opció més suculenta i senzilla a l'hora de fer arribar la imatge al *fragment shader*, es presenta una barrera ineludible que ens obliga a descartar la idea. Per variar, OpenGL ES no suporta textures 3D. La raó que donen per justificar-ho és que els dispositius mòbils estan limitats en recursos, i les textures 3D poden necessitar molta memòria o, al utilitzar-les, poden causar massa *overhead*[29] en el procés de renderitzat.

[29] S'entén per *overhead* s la combinació de l'excés de temps d'execució, memòria o altres recursos com l'ample de banda, que són requerits per assolir un determinat objectiu.

3.2.7.3 Array Textures

Un altre tipus interessant de textures són les *array textures*. Una *array texture* conté sèries d'imatges en 1D o 2D de la mateixa mida. Són similars a les textures 3D.

Van ser ideades perquè a OpenGL previ al 3.0, hi havia un gran cost en el canvi de textura usada (només es pot tenir una sola textura activa cada moment), amb el que es valoraven molt els atles de textures, que permetien reduir l'*overhead* d'aquest canvi entre textures actives.

No es poden utilitzar amb el *fixed pipeline*, però això a nosaltres "tant ens fa", ja que estem obligats a utilitzar *shaders*.

En el nostre cas, podríem utilitzar la *texture array* per passar la imatge fins el *shader*, i allà tractar cada tall de la imatge com si es tractés d'una textura separada (com un atles de textures).

Problema

Igual que tantes altres coses que hem anat veient, les *array textures* no són suportades per l'OpenGL ES. No queda més remei que descartar-les.

3.2.7.4 Buffer Texture

Una *buffer texture* és una textura unidimensional que utilitza com a mètode d'emmagatzematge de les dades de la textura un dels ja explicats *buffer object*.

S'utilitzen per permetre que un *shader* accedeixi a una gran taula (porció) de memòria que és gestionada per un *buffer object*.

La definició d'una *buffer texture* no és molt diferent a la definició d'un altre tipus de textura. La diferència recau en com definir d'on vénen les dades de la textura. Quan normalment cal cridar funcions del tipus *glTexImage* per proporcionar les dades a una textura (un dels paràmetres de la funció és el punter a aquestes dades), en el cas d'una *buffer texture* les dades són a un buffer ja existent, un *buffer object*.

Els altres tipus de textures amaguen la representació binària de les dades de la imatge que contenent. En aquest cas, però, al utilitzar un buffer, això no és possible. L'usuari està limitat a uns tipus d'imatges que existeixen. De totes maneres, hi ha varietat suficient com perquè puguem representar la nostra imatge: el tipus *GL_R8* mateix, representa només la component de color R (vermell) i es representa mitjançant un *unsigned byte*, que equival a *unsigned char*, el tipus amb el que la llibreria ITK ha llegit la imatge mèdica. Podríem utilitzar aquesta component vermella com si de luminància es tractés. Com que s'accedeix al a textura des del *shader*, en el fons tant fa si aquest valor de luminància està repartit entre els 3 components RGB o només en un d'ells, el que importa és que com a mínim un el tingui.

Semblava doncs que podríem utilitzar un *buffer object* per emmagatzemar tota la imatge ITK llegida i llavors utilitzar una *buffer texture* per accedir a ella des del *shader*.

Problema

Un cop més, aquesta funcionalitat no està disponible per a OpenGL ES, així que una opció més que es va haver de descartar.

3.2.7.5 Múltiples textures 2D

Una textura 2D és un objecte que conté una imatge, com el nom indica, en dues dimensions. Es sol aplicar a un polígon de manera que la textura el cobreixi enterament, i així donant-li un nivell de detall a l'objecte representat més alt que si només es fessin servir geometria i diferents colors.

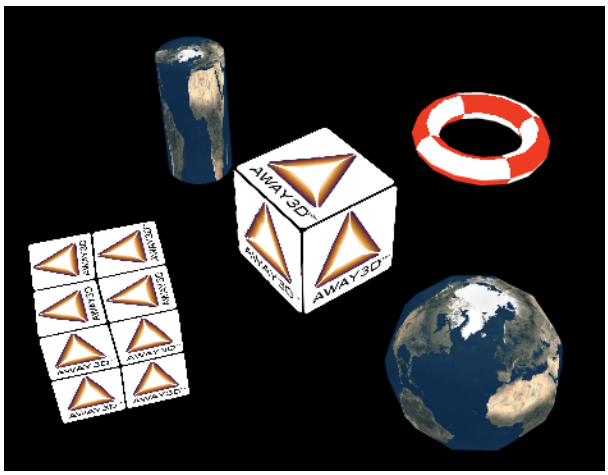


Figura 3.2.7.5.1: Alguns polígons texturats amb textures 2D.

El que ens vam proposar ara va ser veure si era possible utilitzar una o vàries textures 2D per representar la imatge en 3D.

Tipus de les dades a l'interior de la textura

Una textura en dues dimensions pot contenir diferents valors, llistats en la següent taula. Els valors de la columna de l'esquerra simbolitzen què representen les dades que es volen utilitzar com a textura. La columna de la dreta explica què fa OpenGL per tal de normalitzar en un sol tipus de dades, RGBA, els diferents tipus de textures.

Tipus de textura	Significat
GL_ALPHA	Cada element és simplement un component alpha. OpenGL convertirà el valor a *float* i crearà a partir d'ell un element RGBA, on RGB tindran valor 0 i A tindrà el valor de l'alpha.
GL_RGB	Cada element està format per la tripleta RGB. OpenGL ho convertirà al format RGBA, assignant 1 com a valor d'A.
GL_RGBA	Cada element està format per elements del tipus RGBA. OpenGL en aquest cas no ha de fer res.
GL_LUMINANCE	Cada element conté un valor de luminància. OpenGL convertirà el valor a *float* i crearà a partir d'ell un element RGBA on RGB tindran el valor de la luminància replicat i A valdrà 1.
GL_LUMINANCE_ALPHA	Cada element està format per una parella de luminància i alpha. OpenGL el que fa per

crear l'element RGBA és agafar el valor de l'alpha tal qual, i replicar el valor de la luminància en els tres canals de color RGB.

Quina seria, doncs, el millor tipus de dades per passar com a textura al *shader*?

Evidentment, en el cas ideal, seria el color directament en format RGBA, on a sobre la component *alpha* facilitaria el tractament de les transparències en la imatge, alhora de fer-la 3D.

Malauradament, però, recordem que la imatge llegida amb la llibreria ITK es va llegir de tal manera que cada punt de la imatge contenia directament un valor de luminància. Seria ideal si com a mínim, a part d'aquest valor, poguéssim tenir un valor d'alpha que marqués aquelles zones de la imatge que són transparents, i que per tant, des del *fragment shader* es poguessin ignorar (utilitzant doncs el tipus de dades *GL_LUMINANCE_ALPHA*).

Però això no és possible, la imatge ITK llegida està representant cada punt com a un simple *unsigned char* amb valor de luminància. Ja havíem descartat modificar aquesta representació anteriorment, per tant havíem de treballar amb el que disposàvem.

Així doncs, semblava que només hi havia un tipus de textura vàlida: el tipus *GL_LUMINANCE*. Aquí és on es justifica que el tipus de dades de la imatge sigui *unsigned char* i no *char* simplement: la funció per definir la textura provocava un error quan s'indicava que el tipus de la imatge era *char* simplement.

Mides de les textures

Un cop tenim el tipus de dada vàlid, la següent pregunta és: com distribuir la imatge en textures 2D? Doncs bé, la idea seria fer quelcom equivalent a la següent imatge:

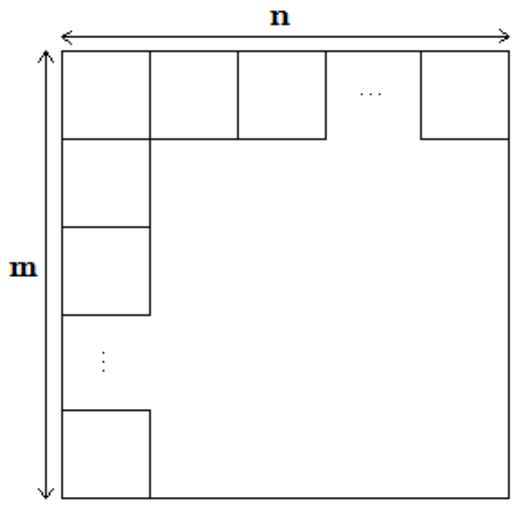


Figura 3.2.7.4.2: Distribució dels talls de la imatge mèdica en una textura 2D.

Però quant valen *n* i *m*? Són dades que cal conèixer abans de començar a plantejar-se les textures 2D com a una opció 100% viable.

Per sort, existeix una funció que ens pot indicar quina és la mida màxima del costat d'una textura en dues dimensions: *glGetIntegerv*.

Així doncs, utilitzant *glGetIntegerv* amb el paràmetre *GL_MAX_TEXTURE_SIZE* vam obtenir el següent resultat:

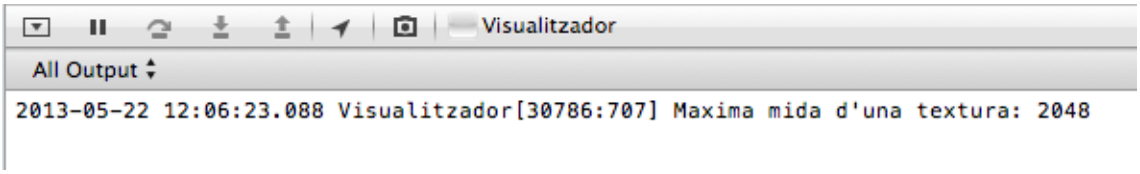


Figura 3.2.7.4.3: Sortida de l'Xcode amb la mida màxima d'una textura.

Aquest valor indica que la mida màxima d'una textura 2D, en aquest dispositiu (iPod), és de 2048. Però 2048 què? 2048 bytes. Això vol dir que si cada element de la textura ocupés més d'un byte, caldria dividir aquest valor 2048 per la mida d'un element de la textura. En el nostre cas, com que la mida d'un *unsigned char* és 1 byte, la mida màxima és 2048 tal qual.

Bé, això volia dir que en una sola textura 2D podíem passar a un *shader* fins a 2048x2048 = 4.194.304 bytes.

Si recordem, la imatge ITK a memòria ocupa (256x256x139) 9.109.504 bytes. Resulta més que obvi que amb una sola textura 2D és impossible passar tota la imatge ITK al *shader*, en faria falta més d'una.

Múltiples samplers

Cada textura està relacionada amb un objecte *sampler* al qual es pot accedir des del *fragment shader*. No hi ha cap impediment que faci que no es pugui utilitzar més d'una textura en aquest *shader*. El que cal fer és enllaçar cada textura amb un objecte *sampler* diferent.

Ara bé, el nombre de textures que es poden enllaçar no és infinit.

Per determinar fins a quantes textures podíem utilitzar, en el dispositiu concret (iPod), es va utilitzar la mateixa funció d'abans: *glGetIntegerv*, aquest cop amb paràmetre *GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS*.

El resultat va ser:

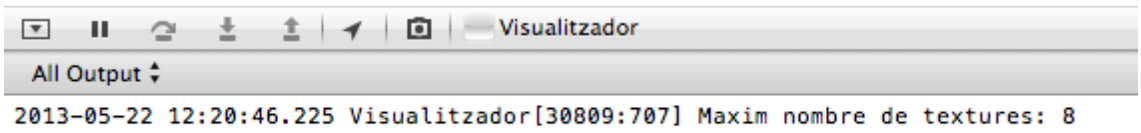


Figura 3.2.7.4.3: Sortida de l'Xcode amb el màxim nombre de textures.

Sembla doncs que podíem utilitzar fins a 8 textures.

És a dir, en total disposàvem de:

$$8 * 2048 * 2048 = 33.554.432\ Bytes\ de\ textura$$

Més que de sobres per poder passar les aproximadament 9MB d'imatge ITK que teníem.

Semblava ser doncs, que havíem trobat un possible mètode amb el qual es podria passar la imatge ITK fins al *shader*.

Com està formada la imatge

Una cosa a tenir en compte, per tal de poder passar la imatge ITK al *shader* amb el format descrit a la figura 3.2.7.4.2, és com està formada aquesta imatge, com està guardada a memòria.

Sabíem les dimensions que té, però no sabíem amb quin mètode es guarda a memòria: Una successió de plans perpendiculars a l'eix X? Una successió de plans perpendiculars a l'eix Y? Una successió de plans perpendiculars a l'eix Z? O d'alguna altra manera totalment diferent d'aquestes?

Necessitàvem conèixer aquesta informació per poder calcular el nombre de talls de la imatge que podíem passar en una sola textura. No és el mateix que cada tall faci 256x256 a que un tall faci 256x139 o 139x256.

Si la imatge fos totalment cúbica ens hagués sigut igual com està guardada, però per desgràcia no ho és.

Per a fer aquest càlcul es van utilitzar dues eines: la funció *getPixel* que proporciona ITK, que donats 3 índexs x, y i z proporciona el valor de la imatge en aquell punt, i la comanda de Linux *od*[30], que bàsicament mostra per consola el contingut d'un fitxer.

El procés que es va seguir per determinar com estava tallada la imatge ITK va ser el següent:

- Primer, es va marcar com a hipòtesi que la imatge estava dividida en talls perpendiculars a l'eix Z. Això implicava que els primers 256*256*2 bytes equivalien a un tall en l'eix Z. També es va incloure en la hipòtesi el pensar que el primer tall seria aquell amb índex de z = 0, cosa gens absurda de pensar.
- Utilitzant la comanda *od -v -s -N512 image01.raw | tr -s ' '*, es van mostrar els primers 512 bytes del fitxer de la imatge. En mostràvem 512 perquè, tot i que nosaltres havíem llegit la imatge com si estigués formada per *usigned chars*, recordem que en realitat estava formada per *shorts*, per tant al mostrar 512 bytes en realitat mostràvem 256 elements.

[30] Més informació d'aquesta comanda a la referència extra [13]

Segons la nostra hipòtesis, aquests bytes haurien de contenir els valors d'una primera fila de punts en el pla XY. La sortida per consola es mostra a la figura 3.2.7.4.5.

- Llavors, utilitzant un simple càlcul d'adreces (les adreces de la figura estan en sistema octal), es va mirar si el valor d'una adreça determinada coincidia amb el valor del píxel que s'esperava que fos allà, degut a la hipòtesis. Això es va fer amb la funció *getPixel*.
- Quan això es complia, es relacionava adreça de memòria amb píxel. Per assegurar-se, es miraven també píxels del voltant, per assegurar-se que realment una adreça corresponia a un píxel, i no que s'havia trobat un altre píxel amb el mateix valor, per casualitat.

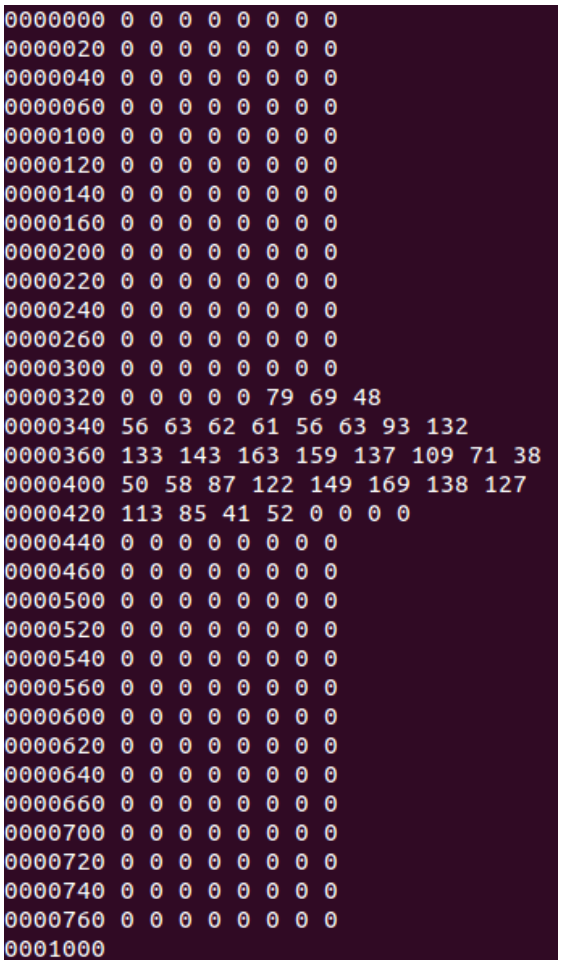


Figura 3.2.7.4.5: Resultat d'executar la comanda od -v -s -N512 image01.raw | tr -s ' '.

Després de seguir aquest mètode pels primers 512 bytes, semblava que la imatge efectivament estava dividida en talls perpendiculars a l'eix Z.

Per a comprovar-ho encara més, es van seguir els passos 2 i 3 pels següents 512 bytes i es va comprovar que efectivament corresponien a un increment en l'eix Y (l'index y dels píxels valia 1), els píxels corresponien a una segona fila de punts en el pla XY.

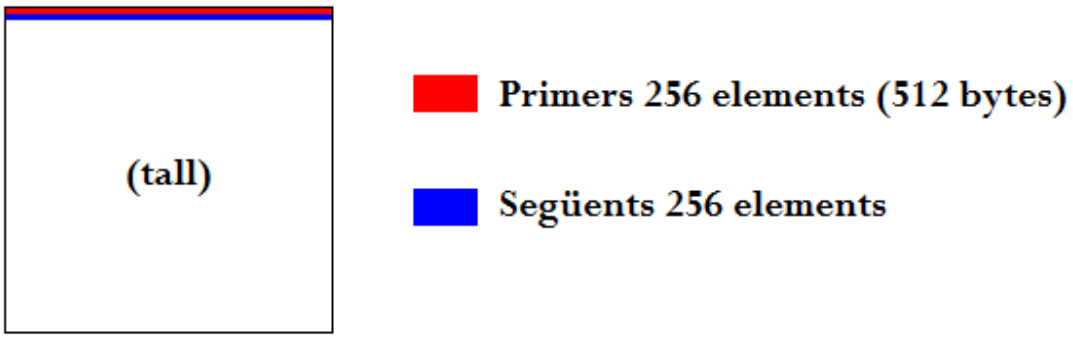


Figura 3.2.7.4.4: En vermell, els primers bytes mostrats amb la comanda od. En blau, els següents.

En la figura anterior es mostren les dues execucions consecutives de la comanda *od*. Segons la nostra hipòtesis, aquest tall hauria de ser un tall perperndicular a l'eix Z.

Per acabar-ho de sentenciar, es van tornar a aplicar els passos 2 i 3 pels primers 512 bytes del que hauria de ser un altre pla XY, és a dir, un altre tall perpendicular a l'eix Z. Per a fer-ho es van imprimir els bytes que començaven a la posició 256*256*2 (recordem que el multiplicar per 2 és pel tema dels elements que són *shorts*) i efectivament corresponien als bytes que s'esperaven.

Així es va concloure que la imatge estava dividida en talls perpendiculars a l'eix Z.

La conseqüència directa d'això va ser deduir els valors d' n i m de la figura 3.2.7.4.2:

Sabent que la imatge es dividia en 139 talls de 256x256 ens va permetre deduir que:

$$n = m = \frac{2048}{256} = 8$$

Això vol dir que en una sola textura cabien 64 talls de la imatge ITK: 8 files de 8 talls cada una.

Aquí és on va fer més servei la simplificació de reduir els talls en l'eix Z de 139 a 128, ja que d'aquesta manera amb només 2 textures 2D teníem prou per passar tota la imatge ITK al *fragment shader*.

3.2.8 Accés a la textura 2D

El problema que ens va venir ara és com accedir a la textura 2D com si es tractés d'una textura 3D en el *fragment shader*.

L'algorisme que es presenta a continuació va ser un dels punts més problemàtics de tot el projecte, ja que com s'ha comentat anteriorment, debugar un *shader* no és gens fàcil. En aquest cas, es fan bastants càlculs matemàtics, que si bé estan formats per operacions senzilles, són complicats d'imaginar conceptualment, amb el que debugar no era fàcil.

La següent funció en GLSL pertany a un *fragment shader* i és responsable de, donades unes coordenades de textura en tres dimensions, accedir a la textura 2D corresponent i retornar-ne el color en el punt indicat:

```
//0
vec4 getColorFromTextures(in vec3 coordinates)
{
	//1
	float xCoord = coordinates.x/xSlices;
	float yCoord = coordinates.y/ySlices;

	//2
	int slice = int(floor(coordinates.z*zSlices));
	int samplerToConsult =
int(floor(float(slice)/float(slicesPerSampler)));

	//3
	if(samplerToConsult == numberSamplers)
	{
		samplerToConsult--;
		slice--;
	}

	//4
	slice = slice - samplerToConsult*slicesPerSampler;

	//5
	float row = floor(float(slice) / xSlices);
	float column = mod(float(slice), xSlices);

	//6
	xCoord = xCoord + column/xSlices;
	yCoord = (row+1.0)/ySlices - yCoord;

	//7
	if(samplerToConsult == 0)
		return texture2D(sampler0, vec2(xCoord,yCoord));
	else if(samplerToConsult == 1)
		return texture2D(sampler1, vec2(xCoord,yCoord));
	else return vec4(0.0);
}
```

Comentem la funció pas a pas:

0: La definició de la funció. A la funció li arriba un paràmetre que és un vector de tres components, que es considerarà que conté coordenades de textura en l'espai 3D.

Com a variables definides globalment, hi ha:

```
uniform sampler2D sampler0;
uniform sampler2D sampler1;
const float xSlices = 8.0;
const float ySlices = 8.0;
const float zSlices= 128.0;
const int slicesPerSampler = 64;
const int numberSamplers = 2;
```

sampler0: és el *sampler* al que està lligada la textura 0, que conté els primers 64 talls de la imatge ITK.

sampler1: és el *sampler* al que està lligada la textura 1, que conté els següents 64 talls de la imatge ITK. Com que per simplificar hem assumit que hi ha 128 talls en l'eix Z en comptes de 139, amb aquest *sampler* i l'anterior n'hi ha suficient per passar tots els talls de la imatge ITK al *shader*.

xSlices: nombre de talls de la imatge que hi ha un al costat de l'altre, ocupant tota l'amplada de la textura. Són 8 perquè recordem que la textura fa 2048 d'amplada mentre que un tall fa 256.

ySlices: similarment a l'anterior, representa el nombre de talls de la imatge que estan un a sobre de l'altre ocupant tota l'alçada de la textura. Són 8 pel mateix motiu i, per tant, indica que hi ha 64 talls en total en una textura.

zSlices: nombre de talls en l'eix Z. Són 128 deguda la simplificació.

slicesPerSampler: quants talls hi ha en un *sampler*. És una variable redundant, es podria calcular fàcilment multiplicant *xSlices* * *ySlices*. Ajuda a la llegibilitat.

numberSamplers: nombre de *samplers* relacionat amb el nombre de textures necessàries per a emmagatzemar tota la imatge ITK.

Cal dir que pot sobtar que les variables com *xSlices* estiguin posades com a constant, en comptes de ser passades com a variable *uniform*. Això és així perquè, com que s'utilitzava la màquina virtual de Linux per fer les proves amb els diferents *shaders*, al tenir definit cada *shader* en un fitxer a part, modificar aquest *shader* no implicava haver de recompilar tota l'aplicació. Així resultava molt més còmode fer proves.

Com que el projecte no s'ha pogut completar amb èxit, no s'ha acabat extraient aquest tipus de dades "*hardcodejades*", però seria una bona pràctica aplicar-ho un cop funcionés. D'altra manera el codi és totalment dependent de la imatge.

Anem a explicar els diferents blocs de codi:

1: Les coordenades de textura que arriben com a paràmetre poden valer de 0.0 a 1.0, com qualsevol coordenada de textura. Llavors, s'ha d'aconseguir modificar el rang de possibles coordenades, per tal de limitar el seu accés a un sol tall de la imatge.

Donat que hi ha *xSlices* talls en l'eix X, si dividim el valor de la coordenada de textura x per *xSlices* el que aconseguim és justament aquesta reducció del rang:

$$R: [0.0, 1.0] \rightarrow [0.0, 0.125] \ (assumint\ xSlices = 8)$$

El mateix passa amb la coordenada de textura segons l'eix Y.

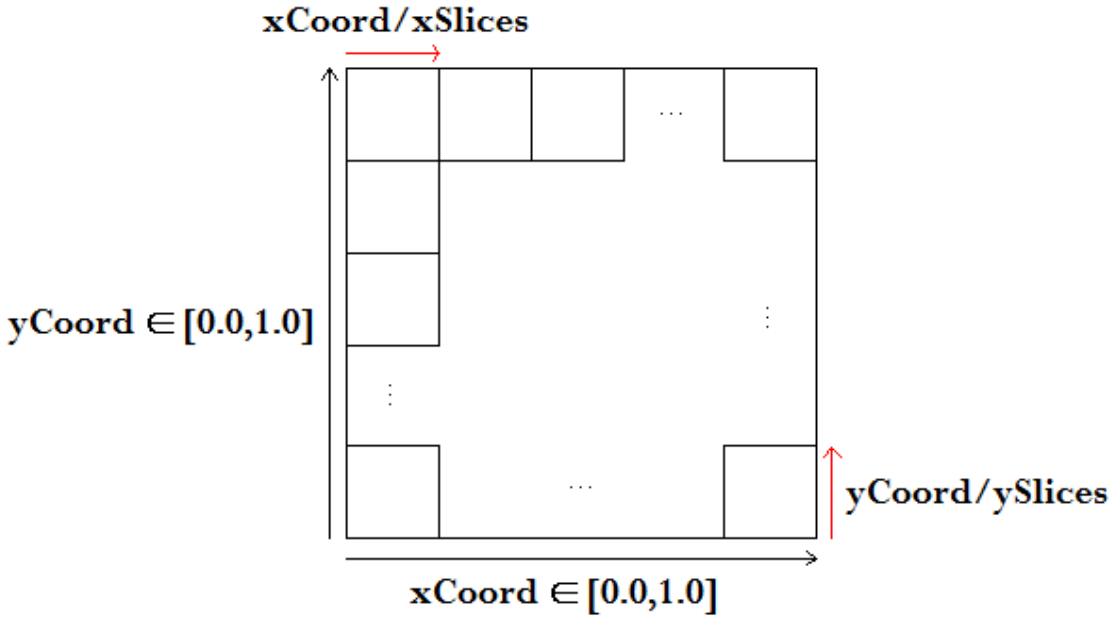


Figura 3.2.8.1: Reducció del rang de les coordenades de textura.

2: Si hi ha *zSlices* talls en l'eix Z, i la coordenada de textura d'aquest mateix eix pot anar de 0.0 a 1.0, al multiplicar la coordenada pel valor de *zSlices* el que obtenim és justament quin tall hem de consultar de les textures, en altres paraules, el que seria l'índex z corresponent de la imatge ITK.

Per exemple:

Assumint *zSlices* = 128:

- Coordenada de textura en eix Z = 1.0 → *slice* = 128
- Coordenada de textura en eix Z = 0.0 → *slice* = 0
- Coordenada de textura en eix Z = 0.5 → *slice* = 64

Cal destacar que la decisió de que al tall 0 li correspongui la coordenada de textura 0 no és arbitrària:

Si s'intercanviés i es fes de tal manera que el tall 128 en comptes de coordenada de textura 1.0 tingués coordenada 0.0, el que passaria és que la imatge que mostraria tal com està definit ara i la que es mostraria amb el canvi serien simètriques respecte un pla en els eixos XY. Això és un problema? Doncs probablement en la majoria dels casos no. Però recordem l'objectiu d'aquest projecte: ajudar a metges i pacients a comprendre resultats d'anàlisis mèdics. **No es pot tolerar** un error induït pel fet d'estar mostrant una imatge simètrica a la real, amb efecte mirall. Podria comportar seriosos problemes.

La següent instrucció el que fa és calcular a quin *sampler* està cada tall. Amb els números de la imatge actual, els primers 64 talls són al primer *sampler* i els següents 64 en el segon.

3, 4: L'if s'explica de la següent manera:

Amb el codi del bloc 2, el que té coordenada de textura segons l'eix Z = 1.0 correspon a *slice = 128* i *samplerToConsult* = 2.

Com que només hi ha 2 *samplers*, el 0 i l'1, el que cal fer és reduir el valor de *samplerToConsult* per evitar *overflow* (intentar accedir a un *sampler* inexistent).

De manera semblant, el valor d'*slice* no pot ser 128. No pot ser-ho perquè tal com està definit tot, a cada textura enllaçada amb un *sampler* hi ha 64 talls de la imatge ITK, 64 talls que estan numerats del 0 al 63 (decisió nostra, no hauria de perquè ser així però facilita els càlculs). Així doncs, quan un *slice* val 64, el que passa és que en realitat es tracta del tall 0 del següent *sampler*. Però com que el cas especial de l'últim tall ens intenta accedir a un *sampler* inexistent (el 2), i hem hagut de reduir el *sampler* a consultar, també hem de reduir el valor de l'*slice* que intenta consultar en 1 unitat, per evitar *overflow*.

5: Un cop tenim calculat l'*slice* dins del *sampler* corresponent, el que es calcula és a quina fila (*row*) i a quina columna (*column*) correspon aquest *slice*, dins la graella de 8x8 que és tota la imatge. Es considera que les files i columnes van de la següent manera:

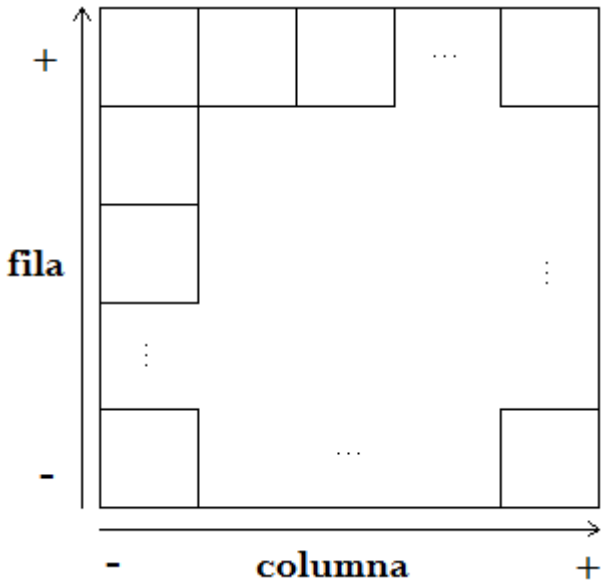


Figura 3.2.8.2: Distribució de les files i columnes de talls de la imatge dins la textura.

6: Amb aquest càlcul les coordenades de textura finalment apuntaran a l'interior de l'*slice* que toca.

És important senyalar una cosa respecte les coordenades de textura en l'eix Y:

OpenGL defineix l'eix Y de les coordenades de textura de la manera que es mostra a la figura següent:

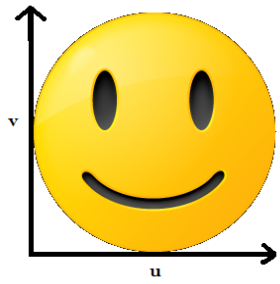


Figura 3.2.8.3: Coordenades de textura. L'eix u equival a l'eix x i l'eix v equival a l'eix y.

Tot i així, iOS defineix a Core Graphics[31] un sistema gràfic de coordenades tal que l'eix Y té l'origen a la part superior de la pantalla i va en direcció a la part inferior de la pantalla. Això el que suposa és que, utilitzant el sistema de coordenades de textura d'OpenGL tal qual, la imatge es vegi invertida.

[31] El Core Graphics Framework és una API basada en C que proporciona renderitzat 2D a baix nivell. S'utilitza per manejar transformacions, color, renderitzat fora de pantalla, imatges, etc.

Així doncs, hi ha dues possibles solucions. Una és fer quelcom com la figura següent:

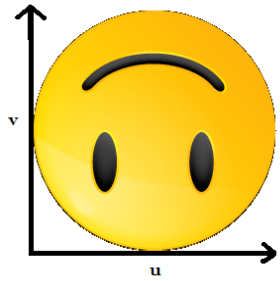


Figura 3.2.8.4: Textura invertida manualment.

Es tractaria d'invertir la textura (la imatge) i seguir utilitzant el mateix sistema de coordenades de textura d'OpenGL, com fins ara.

Això, però, no és possible en el nostre cas, ja que la imatge que utilitzem de textura són un seguit de talls de la imatge ITK, no podem invertir-los un a un.

L'altra solució, és doncs, obtenir la informació de color de la textura al revés per l'eix Y. En comptes de començar per 0.0 fins a 1.0, es comença d'1.0 fins a 0.0. Aquest és el motiu pel qual la segona instrucció del bloc 6 el que fa és apuntar a una fila superior a la calculada i a partir d'allà anar "baixant" segons marqui la coordenada de textura en l'eix v (y).

7: Un cop calculat tot, només queda accedir al *sampler* corresponent i retornar el color que indiqui segons les coordenades calculades.

3.2.8.1 Problemes de precisió

La tècnica explicada anteriorment pot semblar útil, però té un problema:

Com tot allò que treballa amb *floats*, la precisió a l'hora de representar aquests pot causar efectes no desitjats.

Què passa quan una coordenada de textura recau en l'extrem que separa dos talls? Doncs que per problemes de precisió poden passar dues coses:

O bé s'està accedint al tall correcte, o bé:

- Intentant accedir a l'últim texel[32] d'un tall, s'accedeix errròniament al primer texel del tall següent.
- Intentant accedir al primer texel d'un tall, s'accedeix errròniament a l'últim texel de l'anterior tall.

Amb l'exemple gràfic que es mostra a continuació s'entendrà millor el que s'està mirant d'explicar:

Definint manualment la textura:

```
// Cada 4 elements representen un valor RGBA
GLfloat colors[32] = {  0.0, 0.0, 1.0, 1.0, //blau
1.0, 1.0, 0.0, 1.0, //groc
1.0, 0.0, 0.0, 1.0, //vermell
0.0, 1.0, 0.0, 1.0, //verd

1.0, 0.0, 1.0, 1.0, //magenta
0.0, 1.0, 1.0, 1.0, //cyan
0.5, 0.5, 0.5, 1.0, //gris
0.95, 0.65, 0.0, 1.0  //taronja
};
```

I passant-la al *shader* tota sencera en una mateixa textura del format *GL_RGBA*, el que es fa és accedir a aquesta textura de la següent manera:

```
highp vec2 coords = textureVarying.xy;
coords.x /= 2.0;

if (textureVarying.z < 0.5)
{
    coords.x += 0.5;
}
gl_FragColor = texture2D(sampler0, coords);
```

Bàsicament es tracta la primera meitat de la textura segons les coordenades u (eix X) com si fos la primera meitat del cub i la segona part com si fos la segona part del cub.

---

[32] Un texel o *texture element* (o *texture pixel*) és la unitat fonamental en l'espai de textura. Una textura està representada per un conjunt de texels, igual que una imatge està representada per un conjunt de píxels.

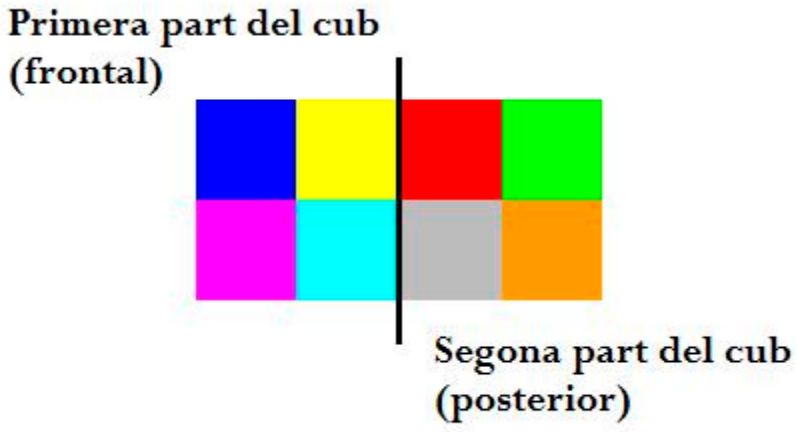
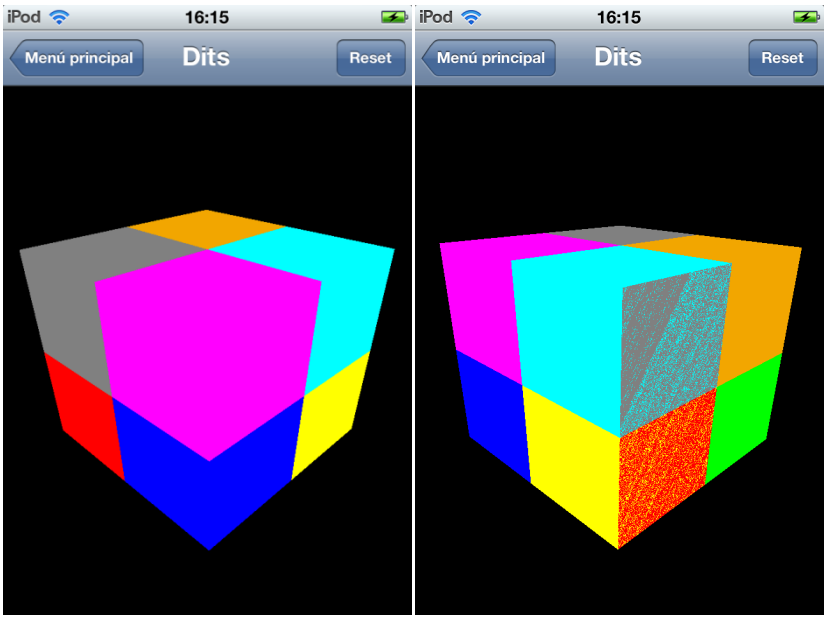


Figura 3.2.8.1.1: Representació visual de la textura definida.

I el que passa al executar és:



Figures 3.2.8.1.2 i 3.2.8.1.3: La primera és un cub amb la textura aplicada i semi rotat cap a la dreta. La segona és el mateix cub semi rotat en sentit contrari.

Aquí es pot veure de manifest l'efecte explicat anteriorment de que el sistema de coordenades de textura segons OpenGL i segons iOS difereixen, per això els colors estan invertits en l'eix Y.

A més, es pot veure que es formen errors de precisió que comporten una barreja de varis colors: aquells que es troben a la divisió dels 2 talls.

Si amb un exemple tant senzill ja passa, ens pot afectar a nosaltres, amb l'algorisme que havíem explicat?

Doncs sí, pot passar. Ara bé, aquest problema, com és evident, només passa als extrems dels talls. Això comporta que, mentre cap tall de la imatge tingui un punt significatiu a l'extrem del quadrat que la representa, aquests problemes de precisió no ens afecten, ja que tant fa agafar una part de la "zona negra" d'un tall com d'un altre.

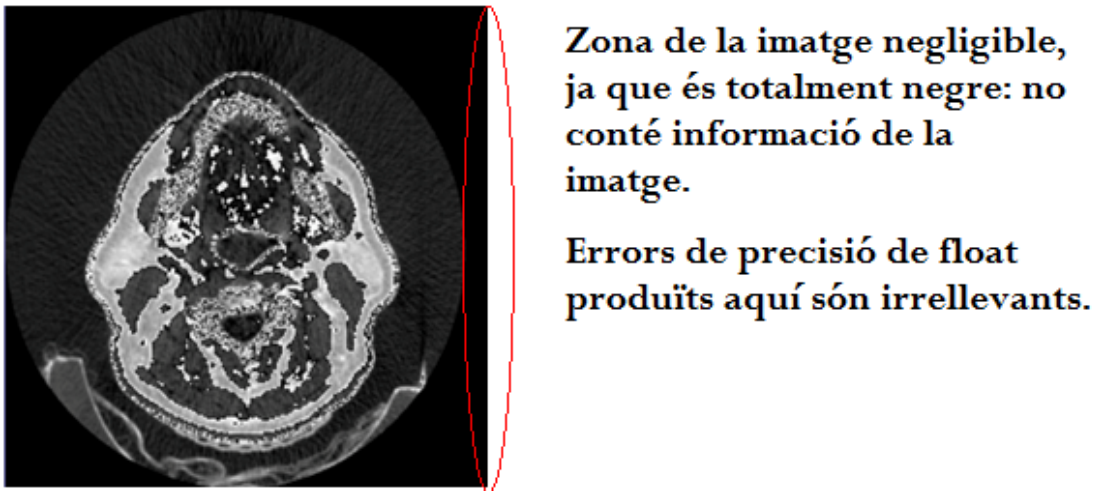


Figura 3.2.8.1.4: imatge en que es pot veure com cap punt significatiu arriba a l'extrem del tall (a la part inferior hi ha una part d'imatge, però no correspon al pacient).

3.2.9 Tècniques de visualització

L'anterior secció parlava de com passar la imatge ITK fins els *shaders*. Però, i com aplicar la imatge un cop passada?

Anteriorment havíem definit un cub com a geometria. Però era la única opció? Doncs realment no.

A continuació es descriuen unes estratègies que es van pensar i començar a provar, i que es van acabar descartant, ja sigui per impossibles, menys eficients o perquè directament es desconeixia el mètode per emprar-les.

3.2.9.1 Un pla per tall

La idea seria definir un pla per cada tall de la imatge ITK , és a dir, si aquesta té 139 talls sobre l'eix Z, doncs definir 139 plans. Llavors, a cada pla assignar-li una textura 2D que contingués únicament un dels talls de la imatge.

Per aconseguir cada un dels talls de la imatge per separat, es va construir una funció anomenada *getSlice(index)*, on el paràmetre indicava quin tall de la imatge ITK es volia.

Aquesta funció bàsicament aprofitava el coneixement adquirit de com estava formada la imatge (utilitzant comanda *od*) per retornar directament un punter a l'adreça de memòria on hi havia aquell tall.

Caldria definir bé la distància entre els plans, però si s'aconseguís fer bé, i tractant el tema de la transparència des del *fragment shader*, podria ser una solució.

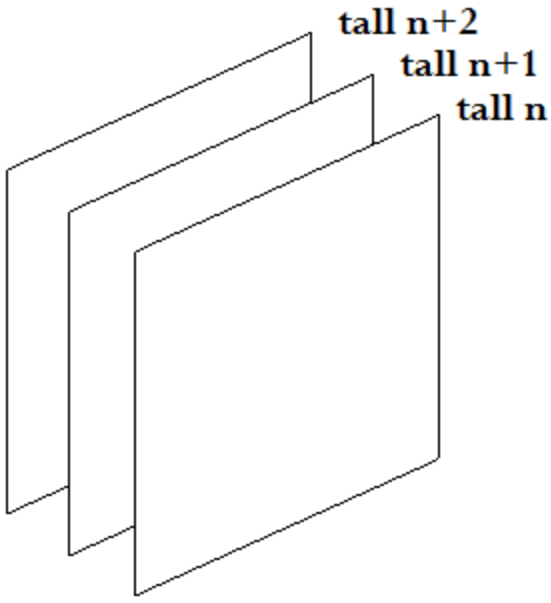


Figura 3.2.9.1.1: Diferents plans equidistants on cada un tindria un tall de la imatge.

Problemes

Resulta inviable, pel fet de tenir un nombre indefinit a priori de talls. Per a cada tall s'hauria de definir uns nous vèrtexs (4) i noves cares (2, recordem que tot i ser plans quadrats s'han d'utilitzar dos triangles per representar-los).

Però el problema no només seria definir dinàmicament les primitives, seria que resulta impossible utilitzar 139 textures diferents (una per pla). No hi ha tants *texture objects* disponibles: com hem vist a la figura 3.2.7.4.3, n'hi ha 8 de disponibles. Per tant, com a màxim es podrien renderitzar alhora 8 d'aquests plans. Això voldria dir haver de recórrer a bucles dins la funció de render (caldria fer 139/8 ≈ 18 iteracions utilitzant les 8 textures) i això podria afectar massa negativament el rendiment de l'aplicació.

A més, ja s'havia comentat que el canvi de textura activa és considerat un canvi costós, i 139 canvis de textura activa no són pocs.

3.2.9.2 Múltiples passades amb un mateix pla

Abans de descriure la idea en sí, cal que definim la funcionalitat d'OpenGL *blending*.

Blending

Després de que els *fragment shaders* enviïn el color a pintar calculat, hi ha un altra procés encara anomenat *blending*.

Aquest procés obté els colors resultants del *fragment shader* i els combina amb els colors ja existents en els buffers de colors destinació del *fragment shader*.

Per defecte, aquest procés està desactivat, i quan un punt determinat del *buffer* de color ja té un valor, i n'arriba un altre, es decideix quin dels dos s'acabarà representant en un dels posteriors passos en el procés de renderitzat: el *depth test*. Allà es mira quin color correspon a un fragment de la primitiva més propera a l'usuari i finalment s'acaba pintant aquell fragment (bàsicament, si una primitiva opaca està superposada a una altra, s'ha de pintar la que es presenta primer en la direcció de visió usuari-escena, en les zones on estiguin superposades les dues primitives).

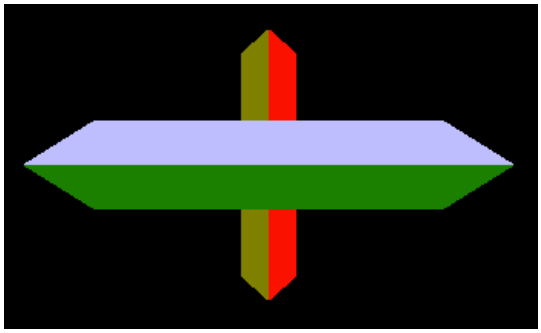


Figura 3.2.9.2.1: Una figura tapa a l'altra gràcies a que el depth test decideix que una està per davant de l'altra.

El *blending* s'utilitza sobretot per simular transparències, ja que permet aquesta barreja de colors del ja present en el *color buffer* i el que ve de sortida d'un

fragment shader: donant més pes a un o altre s'aconsegueix més o menys transparència.

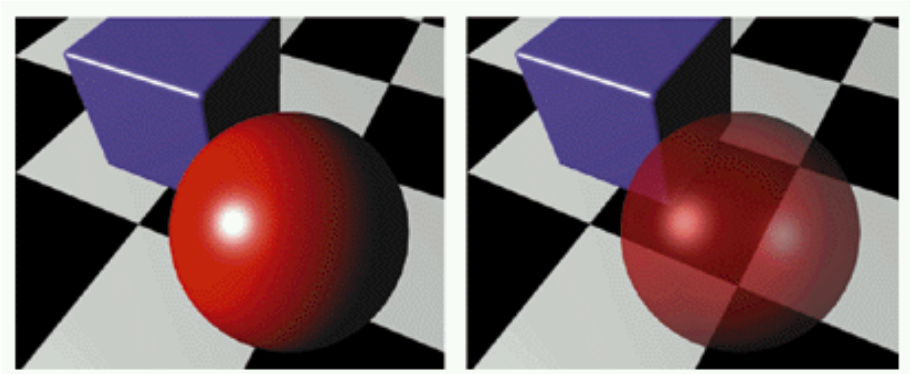


Figura 3.2.9.2.2: A l'esquerra, una imatge sense blending. A la dreta, la mateixa imatge però utilitzant blending per simular la semitransparència de l'esfera.

I com es podria fer servir, llavors, la funció de *blending*? Doncs com es descriu en el següent mètode:

S'utilitzaria un pla, en comptes del cub que havíem definit, com a primitiva a pintar.

Llavors, per cada tall de la imatge ITK, s'enviaria a pintar aquest pla. S'enviaria amb una textura que contindria els valors de color corresponents a aquell tall de la imatge ITK que es vol pintar. Això es faria amb la funció *getSlice* ja descrita en l'anterior apartat.

Més concretament, el que caldria fer seria un bucle a l'interior de la funció de pintar, que bàsicament inicialitzaria una textura de dimensions 256x256 que contindria la informació d'un cert tall de la imatge. Un avantatge respecte a la tècnica proposada en l'anterior apartat és que amb aquest sistema només faria falta un sol pla i un sol objecte textura.

Llavors, aprofitant la funció de *blending*, el que es faria seria donar més pes al color ja present al *color buffer* que no el que vingués nou, pes que relacionaríem amb la profunditat del tall de la imatge ITK que estem pintant. Així, si pintéssim la imatge ITK del tall més proper al més llunyà:

- Si en aquell punt hi havia transparència fins el moment:
  - Un nou punt transparent deixaria transparència igual.
  - Un nou punt no transparent generaria color, però afeblit per la distància del tall.
- Si en aquell punt ja hi havia color no es faria res, hauria de predominar el color del primer tall.

Això es pot aconseguir utilitzant el mètode de fusió *GL_MAX* (màxim entre els dos colors). A més, caldria reduir l'alpha (opacitat) dels talls de la imatge ITK en teoria més allunyats de l'observador, per tal que es pogués complir el segon subpunt del primer punt anterior.

Problema

Un pas de renderitzat per tall de la imatge ITK és excessiu. Amb la imatge que estem fent servir per provar això equival a 139 passades (128 si assumim la simplificació). Provant-ho amb la màquina virtual de Linux anava massa lent com per fins i tot intentar-ho passar a la versió d'iOS.

A més, això el que faria seria mostrar un pla amb la imatge, simulant 3D, però en realitat segueix sent un pla, amb el que les rotacions sobre els eixos X i Y no serien correctes (es veuria un pla de costat i una imatge deformada).

Tot i així es va investigar si seria possible utilitzar rotacions amb la pròpia llibreria ITK, ja que oferia aquest tipus de transformacions en mode de filtres, però aquestes només es podien aplicar en dues dimensions, sobre el pla principal de les imatges.

3.2.9.3 Reconstrucció 3D Multiplanar

Hi ha una tècnica que consisteix en disposar 3 plans ortogonals com a la següent figura:

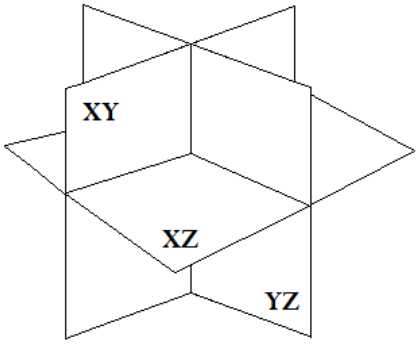


Figura 3.2.9.3.1: Plans amb els que es mostraria la imatge.

Llavors, cada una d'aquestes imatges té la informació corresponent a la imatge mèdica tallada de la manera com indica el pla.

Aquesta solució permet desplaçar el pla en la direcció de la seva normal (vector perpendicular al pla) i així anar mostrant la informació dinàmicament.

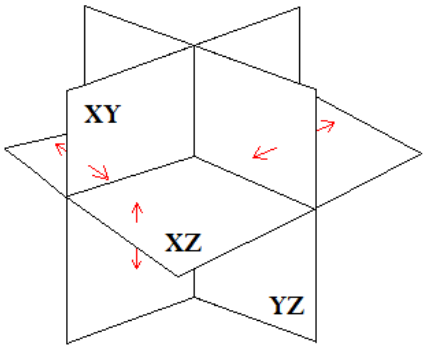


Figura 3.2.9.3.2: Al moure els plans, s'avança/retrocedeix el tall a mostrar.

Problemes

A part de que no es podria disposar fàcilment de la informació visual necessària per representar aquells plans que no siguin l' XY (ja que la imatge ITK està dividida així), aquesta solució no aporta una visualització en 3D com a tal, que és el que es buscava. Per tant, tot i aproximar-s'hi, no serveix.

3.2.9.4 Volume rendering

Aquesta tècnica consisteix en construir volums de la següent manera:

A partir del que seria la posició de la càmera/observador es tracen un seguit de rajos que tenen com a origen aquesta posició de la càmera, i com a destí, cada un dels fragment de la geometria renderitzada, en el nostre cas, el cub. Aquesta tècnica és coneguda amb el nom de *ray casting*.

Llavors, a partir d'aquests rajos, es reconstrueix el volum. Per a fer-ho hi ha vàries opcions, però nosaltres en descrivim dues:

- Es calcula la primera intersecció del raig amb el volum a representar i, un cop trobada, s'agafa el valor del color d'aquell punt. Es valora més una intersecció raig-volum més propera que una de més llunyana, és a dir, el color resultant serà més intens. Si no es fa així, no es podria distingir la profunditat: dos rajos que travessessin el volum acabarien donant el mateix color resultant, per tant, semblaria que els dos punts estiguessin sobre un mateix pla (tall de la imatge) quan en realitat no hauria de perquè ser així.
- Es travessa sempre tot el volum i per cada punt que no és transparent, es calcula la seva aportació en el color final d'aquell punt. Per tant, com més punts estiguin en la línia del raig, més intens serà el color del fragment corresponent.

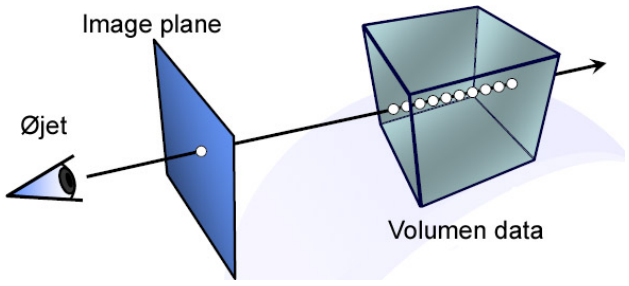


Figura 3.2.9.4.1: Representació visual del raycasting

Sigui quin sigui el mètode aplicat (dels dos anteriors), opcionalment hi ha la possibilitat d'utilitzar una textura d'una dimensió (en el fons, una simple array) que actuï com a *hash*. Aquest *hash* es consultaria utilitzant el color calculat amb el *raycasting*, i aquest es traduiria a un color en format RGBA. D'aquesta manera, s'aconseguiria que la imatge es veiés composada en colors depenent de la distància del punt a l'origen del raig (primer mètode anterior) o de la quantitat de punts amb informació de luminància que ha travessat el raig (segon mètode anterior).

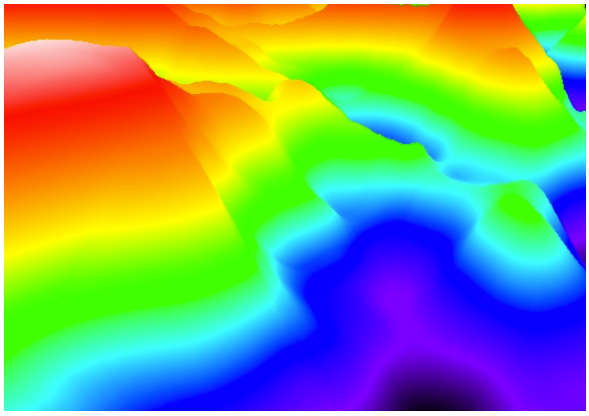


Figura 3.2.9.4.2: Muntanyes pintades utilitzant una textura 1D i accedint-hi depenent de l'alçada.

Sigui quin sigui el mètode escollit, s'aplica per tots els fragments a renderitzar i el resultat és la representació en 3D del volum.

Aquesta va ser la tècnica que es va provar d'implementar durant més temps, ja que semblava la més eficaç i alhora semblava que no hagués de preocupar excessivament l'eficiència.

El primer problema va resultar de com calcular la direcció dels rajos que calia traçar fins a cada fragment.

L'origen dels rajos és la càmera (observador) i la direcció és una per cada fragment (píxel) ocupat per la geometria que estem pintant.

Un mètode per calcular aquestes direccions resulta d'aprofitar-se de les coordenades de textura del cub que ja tenim definit.

Resulta que si primer pintem el cub utilitzant com a color el valor de les coordenades de textura de les cares posteriors, i després el mateix però per les cares frontals, el valor de la resta dels dos colors defineix el la direcció del raig de cada fragment.

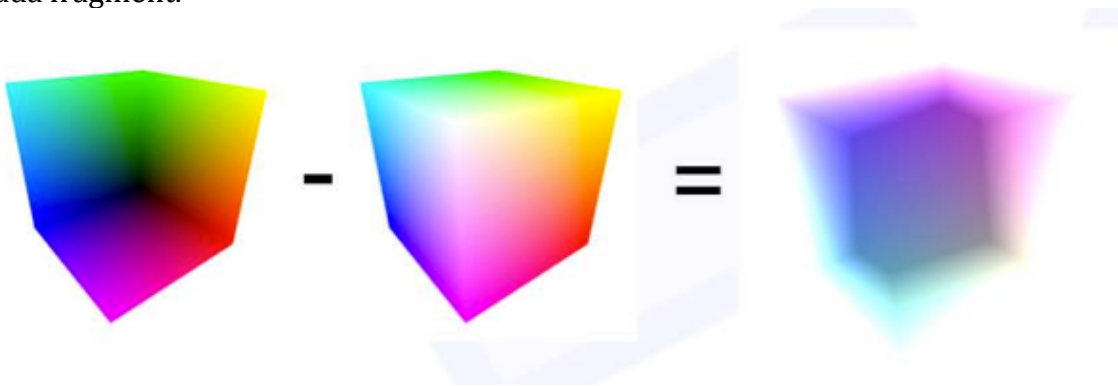


Figura 3.2.9.4.3: A l'esquerra, cares posteriors del cub pintades amb les coordenades de textura. Al centre, el mateix però amb les cares frontals. A la dreta, el valor de la resta, és a dir, representació visual dels rajos càmera-fragment.

Ara bé, per poder aplicar aquest mètode cal que d'alguna manera tinguem accés a les dos versions anteriors, la del cub pintada amb les cares posteriors i la pintada amb les cares frontals, com es podia fer això?

Caldria forçosament fer dos passos de renderitzat (pintar dues vegades el cub). En el primer, es pintaria el cub amb les cares posteriors i utilitzant les coordenades de textura com a color. En el segon, s'utilitzaria el resultat del primer i, aprofitant que ja s'estaria pintant el cub amb les cares frontals, es calcularia la direcció dels rajos i s'aplicaria el *ray casting* (és a dir, en el fons el *volume rendering*) tot en un.

Si això es podia fer, ja teníem mètode per mostrar la imatge en tres dimensions.

El primer va ser descobrir com podíem emmagatzemar el resultat del primer pas de renderitzat. La resposta no va tardar en arribar: guardant-lo en una textura 2D.

Per a fer-ho, calia definir un *framebuffer object*, ja que el que OpenGL utilitza per defecte és el que s'acaba mostrant per pantalla.

Llavors, definint un *framebuffer object* i assignant com a *color buffer* l'espai corresponent a una textura 2D, el que es faria és que quan es pintés amb destinació aquest *framebuffer*, en realitat s'estigués pintant a la textura (la textura hauria de tenir les dimensions del *viewport*, o finestra on es mostra tota l'escena OpenGL).

Un cop fet això, en el segon pas de renderitzat, es podia accedir a la textura creada com si d'una textura més es tractés. L'únic que era una mica "complicat" era, per cada fragment, calcular les coordenades amb les que s'ha d'accedir a la textura. Com que la textura conté tota la finestra de visió, no només el cub, per calcular-les calia transformar el fragment actual al sistema de coordenades de dispositiu:

Donat un fragment amb coordenades de *clipping*, la transformació a coordenades de dispositiu és tant senzilla com:

$$(x_d, y_d) = (x_c/w_c\,, y_c/w_c)$$

On x_d i y_d són les coordenades del fragment en el sistema de coordenades de dispositiu, x_c i y_c són coordenades del fragment en coordenades de *clipping*, i w_c és la quarta coordenada del fragment, anomenada component homogènia[33], en el sistema de coordenades de *clipping*.

Així doncs, un cop solucionat aquest tema, ja estàvem en condicions d'implementar el sistema.

A continuació es descriu el cos del *fragment shader* que intentava implementar *volume rendering* per intentar mostrar la imatge. Utilitza la funció *getColorFromTextures* descrita a la secció 3.2.8.

[33] La coordenada homogènia és aquella que permet aplicar perspectiva a la representació 3D d'una determinada escena. Més informació sobre aquestes coordenades a les referències extres [14] i [15].

```
//0
void main()
{
	//1
	vec3 rayPosition = textureVarying.stp;
	vec2 bfcCoordinates = ((fragmentPosition.xy /
fragmentPosition.w) + 1.0)/2.0;
	vec3 rayDirection = normalize(texture2D(backTexture,
bfcCoordinates).stp - rayPosition);

	//2
	vec4 finalColor = vec4(0.0);
	vec4 sampledColor = getColorFromTextures(rayPosition);
	float intensity = 1.0;
	bool keepSampling = true;

	//3
	if(sampledColor != vec4(0.0, 0.0, 0.0, 1.0))
	{
		keepSampling = false;
		finalColor = sampledColor;
	}

	//4
	while (keepSampling)
	{
		//5
		intensity -= 0.1;
		rayPosition += rayDirection * sampleStep;

		keepSampling = intensity > 0.0 &&
		!any(greaterThan(rayPosition, allOnes)) &&
		!any(lessThan(rayPosition, allZeros));

		//6
		if(keepSampling)
		{
			sampledColor =
			getColorFromTextures(rayPosition);

			if(sampledColor != vec4(0.0, 0.0, 0.0, 1.0))
			{
				finalColor = sampledColor * intensity;
				keepSampling = false;
			}
		}
	}

	//7
	if(finalColor == vec4(0.0)) discard;
	gl_FragColor = finalColor;
}
```

Comentem l'algorisme pas a pas:

0: Com a variables globals, a part de les descrites a la secció 3.2.8, hi ha:

```
varying vec4 fragmentPosition;
uniform sampler2D backTexture;
const vec3 allOnes = vec3(1.0);
const vec3 allZeros = vec3(0.0);
const float sampleStep = 1.0/130.0;
```

fragmentPosition: Són les coordenades del fragment en *clipping space*.

backTexture: *Sampler* que correspon a la textura a la que hem renderitzat en el primer pas, la que conté les cares posteriors del cub pintades utilitzant les coordenades de textura com a color.

allOnes, allZeros: Vectors constants plens d'uns i zeros respectivament. Es fan servir per detectar quan un raig ha travessat completament el cub en alguna de les components (x, y o z), i per tant acabar amb el bucle.

sampleStep: Distància de mostreig, és a dir, distància entre punts del raig càmera – fragment on es comprova si hi ha intersecció amb el volum o no.

1: El primer és inicialitzar el raig. La primera instrucció el que fa és posar com a origen del raig les coordenades de textura. La segona, calcula les coordenades amb les que accedir a la *backTexture*, les coordenades de dispositiu. El motiu de sumar 1.0 i dividir per 2.0 és que les coordenades de dispositiu van de [-1, 1], mentre que les de textura van de [0, 1], amb el que la transformació d'unes a altres és tant senzilla com això, sumar 1 i dividir per 2.

$$f: [-1,1] \rightarrow [0,1]$$

$$f(x,y) = (\frac{x+1}{2}, \frac{y+1}{2})$$

La tercera instrucció simplement accedeix a aquesta la textura i calcula la direcció del raig.

2: Inicialitzem el color final, de moment buit, i comencem a mostrejar ja i a posar el resultat dins de *sampledColor*.

Ajustem la *intensitat* a 1, i a mesura que el raig avanci, anirem reduint la intensitat, perquè el color resultant sigui més feble.

L'última inicialització és la d'un booleà que determinarà quan parar el bucle principal.

3: Si el color que hem mostrejat al inicialitzar *sampledColor* ja és un color vàlid, ja no cal buscar més. Vàlid és tot color menys el negre, ja que la textura és del tipus *GL_LUMINANCE*, i per tant els punts on la luminància valgui 0 tindran assignat el color negre.

4: Bucle principal. Mentre no s'ha trobat cap intersecció amb la imatge que doni un valor no transparent, o mentre no s'hagi travessat completament el volum, o mentre no haguem fet tants passos que la intensitat que donaríem al color ja sigui nul·la, seguim mostrejant en la direcció del raig.

5: La primera instrucció redueix la intensitat del color si el trobéssim en el punt que ara mostrejarem. La segona, fa avançar el punt de mostrejat en la direcció del raig. La tercera determina la condició de sortida.

6: Mostregem. Si el color trobat és diferent de negre (transparent), podem parar el bucle i assignar a *finalColor* el color mostrejat. Si el color és negre, no fem res. Utilitzem *intensity* per afeblir aquells colors que han requerit més passos, ja que vol dir que es troben "més endins" en el volum (seguint la direcció del raig).

7: Finalment, el color final és el color emmagatzemat a *finalColor*. Si no s'ha trobat cap intersecció, directament descartem el fragment.

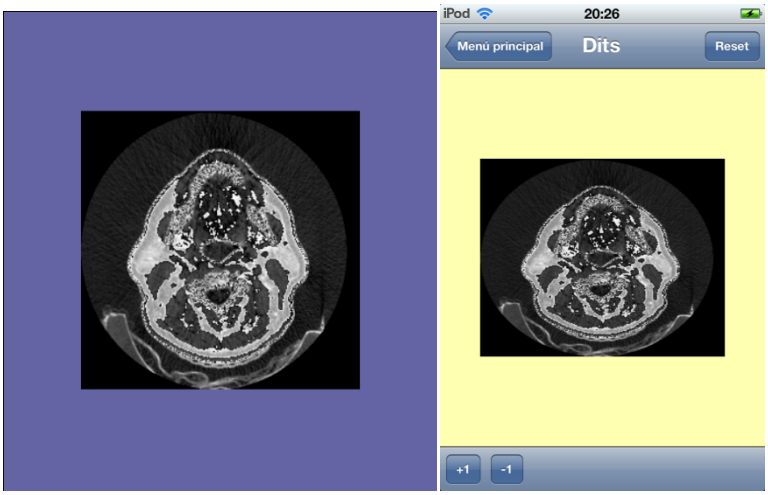
3.2.9.5 Problemes amb el volume rendering

Realment semblava que la tècnica del *volume rendering* havia de servir i que aconseguiríem representar la imatge en 3D. Però no va ser així. Varis problemes[34] van sorgir que van dificultar la seva implementació. A continuació s'expliquen els més greus i amb els que es va acabar.

Per començar, com ja hem explicat anteriorment, els *shaders* es fan per cada fragment, per tant, si fan molts càlculs l'aplicació se'n ressent. Tant és així que l'iPod no es veia capaç d'executar l'aplicació. Per tant, es van haver de desenvolupar enterament utilitzant la màquina virtual de Linux. Sempre pensant, però, que després caldria adaptar-los per l'iPod.

Un altre problema, més conceptual, que va sorgir després de llargues hores sense entendre què passava, va ser que es va descobrir que les textures 2D que contenien la imatge no arribaven bé al *shader*:

Quan a la funció per inicialitzar una textura se li passava com a mida d'aquesta 256x256, és a dir, la mida d'un sol tall de la imatge, i s'utilitzava un *fragment shader* per pintar-la, aquesta sortia bé: s'esperava que sortís un sol tall de la imatge ITK i això és el que es mostrava.



Figures 3.2.9.5.1 i 3.2.9.5.2: Representació d'un sol tall de la imatge tant en Linux com en iOS.

La funció per inicialitzar les dades té la forma:

```
glTexImage2D(par1, par2, par3, amplada, alçada, par6, par7,
par8, punter);
```

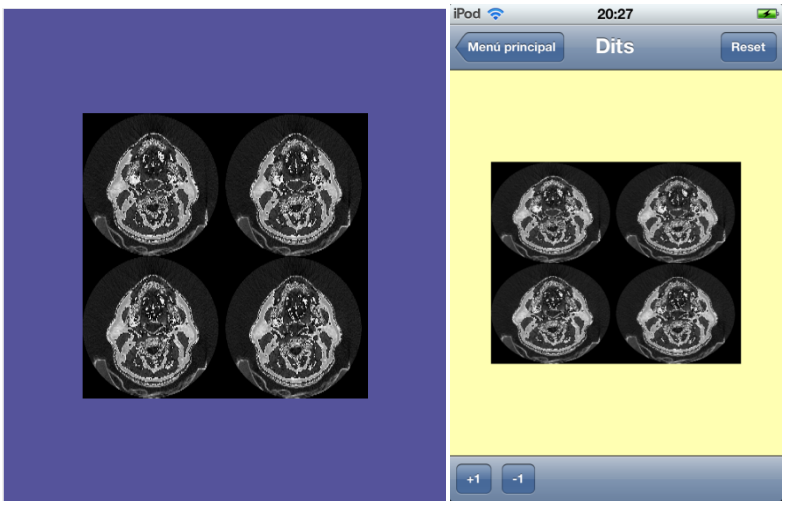
On par{n} són paràmetres sense rellevància pel cas que ens ocupa, alçada i amplada són les mides de la textura i punter és el punter a la textura.

[34] Més detalls a la secció 7.

La qüestió és que substituint la crida de la funció anterior de manera que la nova amplada sigui el doble de l'anterior, és a dir, si la imatge fa 256 vol dir posar 512 com a amplada, el resultat no era lògic.

Pels coneixements de textures que havia adquirit, si la textura no era quadrada, quan es mostrava en un espai quadrat, aquesta es deformava per adoptar-ne la forma. És a dir, en el nostre cas, que tenim un cub (quadrat si no el rotem), si la textura fa el doble d'alçada que d'amplada, el que s'esperaria veure serien dos talls de la imatge, un al costat de l'altra, i "allargades" (el doble d'altes que d'amples).

Però en canvi, el que acabava mostrant-se és el que es pot observar en les següents figures:



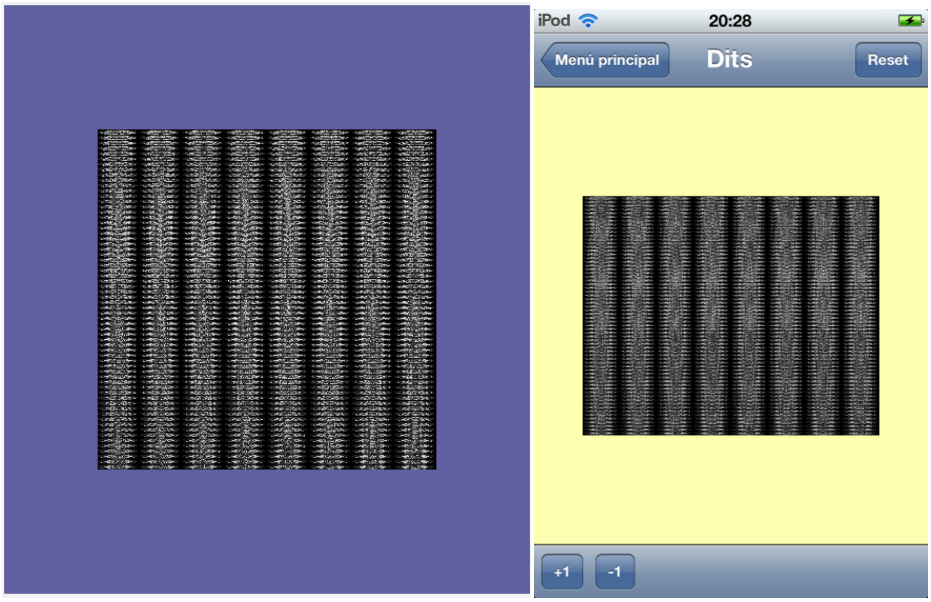
Figures 3.2.9.5.3 i 3.2.9.5.4: Intent de representar dos talls allargats.

Aquí és on va sorgir el primer gran dubte. Què estava passant? Com podia ser que sortissin 4 talls en comptes de 2? Semblava que alguna cosa s'havia entès malament de textures, però no s'acabava de veure el què.

Un cop vist aquest problema, es va decidir pintar el màxim de la textura, 2048x2048, per veure què es mostrava. Per a fer-ho, només calia cridar a l'anterior funció amb 2048 com a alçada i com a amplada.

Amb la teoria a la que havíem arribat, havien de sortir 8x8 = 64 talls de la imatge, és a dir, mitja imatge ITK, sense ni una deformació.

Però, un cop més, el resultat no va ser l'esperat:



Figures 3.2.9.5.5 i 3.2.9.5.6: Intent de representar 64 talls de la imatge ITK.

Tot i que les imatges anteriors són difícils d'apreciar per la limitació de la mida, es pot intuir que 64 imatges no n'hi ha. De fet, n'hi ha **8x64 = 512**. Això va descol·locar totalment.

Per una banda, el problema només era present en l'eix y. Sortien 8 cops més imatges de les esperades.

Per una altra, i el fet més estrany de tots, **sortien 373 talls més dels possibles**. Tenint en compte que la imatge només tenia 139 talls, com era possible que en sortissin 512?

Per molt que es va investigar, no es va trobar resposta a aquestes preguntes.

Es va recórrer a l'ús de fòrums com el conegut *stackoverflow* o al mateix oficial d'OpenGL i, o no es va rebre resposta, o no es va trobar explicació. El problema també residia en què aquests dos fòrums tenien restriccions a l'hora de poder adjuntar imatges i/o urls a posts creats per nous usuaris (acabats de registrar), pel qual a vegades no es tenia permès adjuntar res i, sense tenir una mostra gràfica del que s'intenta explicar, costava molt posar-se en situació als que intentaven respondre.

Fins i tot es va arribar a dubtar si la imatge estava llegida correctament, ja que el tipus original de la imatge era *short* i s'havia llegit *unsigned char* a la brava. Per això mateix es va intentar llegir la imatge amb el seu tipus original i llavors aplicar un filtre d'ITK, el filtre *Cast Image Filter*, que bàsicament transformava el tipus de dada dels punts (de *short* a *unsigned char*, en el nostre cas) però el resultat va ser el mateix.

Així doncs, com a solució barroera, es va recórrer a la modificació del *fragment shader*. La variable *ySlices*, que originalment havíem inicialitzat a 8, ara passaria a valer 64.

El problema, llavors, va ser que al utilitzar el procediment del *ray casting*, com que les imatges estaven exageradament "xafades" (eren 8 cops més amples que altes), la imatge resultant patia masses deformacions:

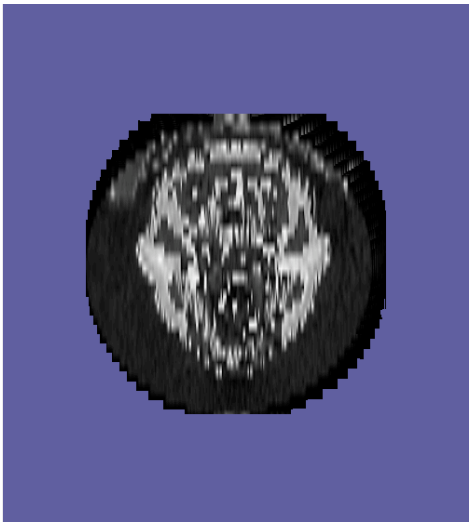
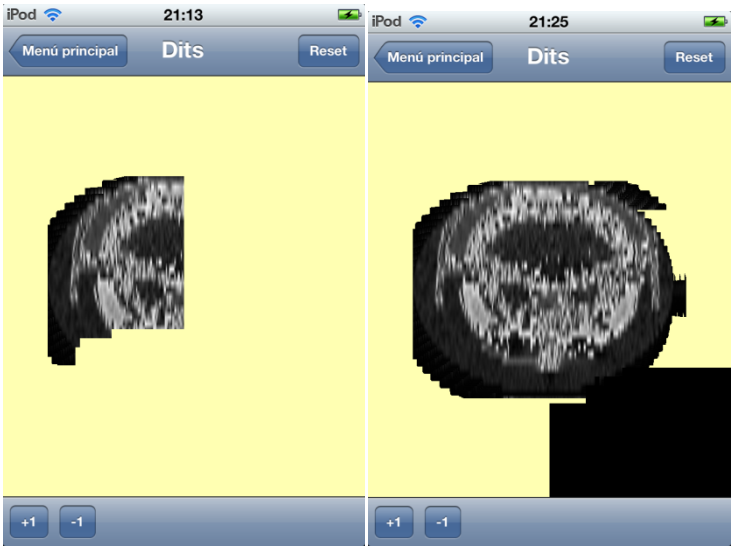


Figura 3.2.9.5.7: Imatge deformada a Linux.

La imatge corresponent a la versió d'iOS no es pot mostrar del tot, ja que **el *shader* era massa lent**, i no es podia executar bé. El resultat era un pampallugueig constant. Com que la pantalla de l'iPod tenia un comportament estrany, amb prou feines es van poder reunir les següents captures per mostrar:



Figures 3.2.9.5.8 i 3.2.9.5.9: Intents de representació en l'iPod Touch.

El més curiós és que intentant optimitzar el *shader* **el comportament variava**. Per exemple, si sabíem que la instrucció *discard* és molt costosa, es podria eliminar l'if del bloc de codi 7. Es podria fer perquè, traient-lo, el color que quedaria del

fragment seria transparent. Tot i així, estranyament, si eliminàvem l'if el resultat era:

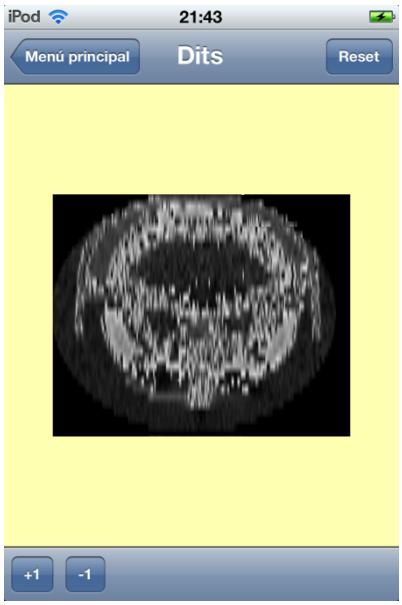


Figura 3.2.9.5.10: Representació utilitzant un shader en teoria optimitzat

Sense cap mena de lògica aparent, les cantonades negres reapareixen amb aquesta optimització. Això sí, el *shader* anava mínimament més ràpid, tot i que amb un resultat incorrecte, realment tant fa.

També es va detectar un altre error de comportament estrany. Aquest error només afectava a l'aplicació a iOS. El codi equivalent en la màquina virtual de Linux funcionava, però en iOS tenia un comportament no desitjat.

Es tracta del mètode que havíem comentat de renderitzar primer a una textura i llavors, en un segon pas de renderitzat, utilitzar aquesta mateixa textura per calcular la direcció de sortida dels rajos que traçaríem.

Resulta que, si en comptes d'accedir a la textura i agafar el valor per calcular la sortida d'un raig, pintàvem directament el valor agafat, el resultat a Linux era el següent:

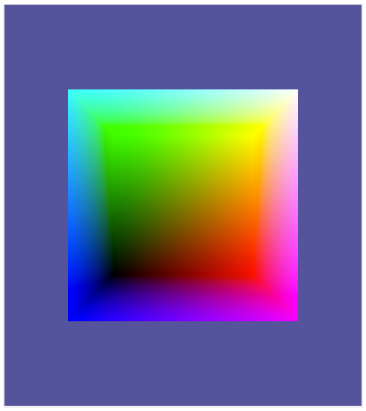


Figura 3.2.9.5.11: Cub pintat amb les coordenades de textura de les cares posteriors.

Aquesta és la sortida que cabia esperar: un cub pintat amb les coordenades de textura de les cares posteriors. En canvi, exactament el mateix codi però per iOS, el resultat era:

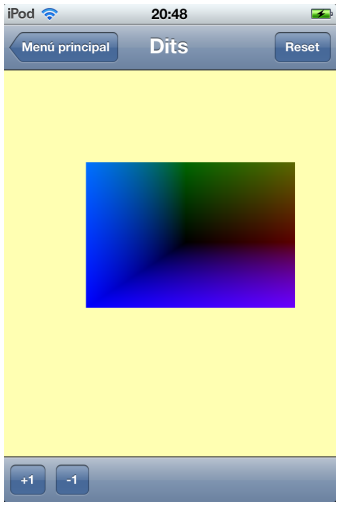


Figura 3.2.9.5.12: Intent de pintar el cub amb les coordenades de textura de les cares posteriors.

Aparentment, el cub està deformat. Però no, en realitat, si entre els dos passos de renderitzat canviem el color de fons el que tenim és:

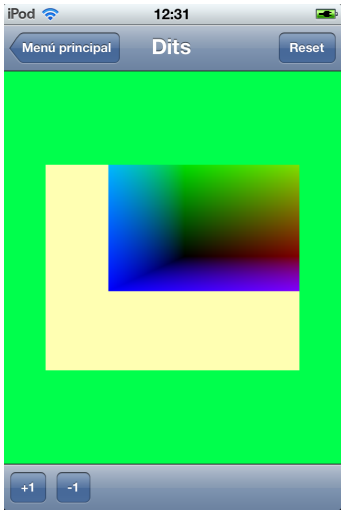


Figura 3.2.9.5.13: El mateix que la figura anterior però canviant el color de fons en el segon pas de renderitzat.

La part groga és doncs part del primer pas de renderitzat, és a dir, part de la textura. Això vol dir que a l'hora d'accedir a la textura amb el càlcul explicat en l'anterior algorisme, bloc de codi 1, no es fa correctament. No s'entén que el mateix codi exactament funcioni de dues maneres tant diferents simplement canviant on s'executa. Per molt que es va investigar, no es va trobar solució.

3.2.9.6 Polint la imatge

Deixant de banda els problemes que hagin pogut sorgir amb el *volume rendering*, la imatge que es generava no era ben bé l'esperada: tenia zones fosques, quasi negres, que en principi haurien de ser transparents.

Si ens fixem amb l'algorisme descrit per aplicar el *ray casting*, veurem que això passa perquè es considera un punt com a transparent quan el color que s'obté de la textura amb els talls de la imatge és negre. Només que el color sigui mínimament més clar, ja no es considera transparent, i per tant s'obté com a color final.

Això és clarament un problema, en el sentit que mai podrem avançar més que el primer tall, si quan trobem un color que no és negre, però és molt proper, ja ens aturem.

Per sort això té fàcil solució. Existeixen un parell de filtres ITK que ens poden servir per solucionar això.

Binary Threshold Filter

Aquest filtre transforma una imatge en una imatge binària. Per aplicar-lo es defineixen:

- Valor que s'assignarà als punts dins el llindar: *insideValue*.
- Valor que s'assignarà als punts fora del llindar: *outsideValue*.
- Interval de valors de punts que formen el llindar: [min, max].
    - Si un punt té de valor un valor fora del llindar, se li assigna de valor *outsideValue*.
    - Si un punt té de valor un valor de dins el lindar, se li assigna el valor *insideValue*.

Per valor d'un punt s'entén el valor de la imatge ITK llegida en aquell punt, en el nostre cas, el valor de luminància.

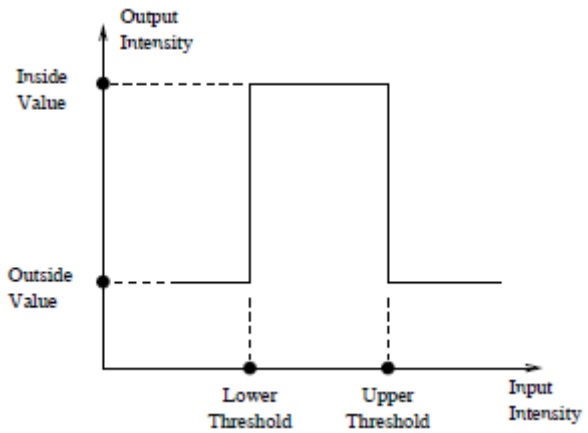


Figura 3.2.9.6.1: Imatge que descriu el procés del filtre.

Un exemple de com queda una imatge després d'aplicar aquest filtre:

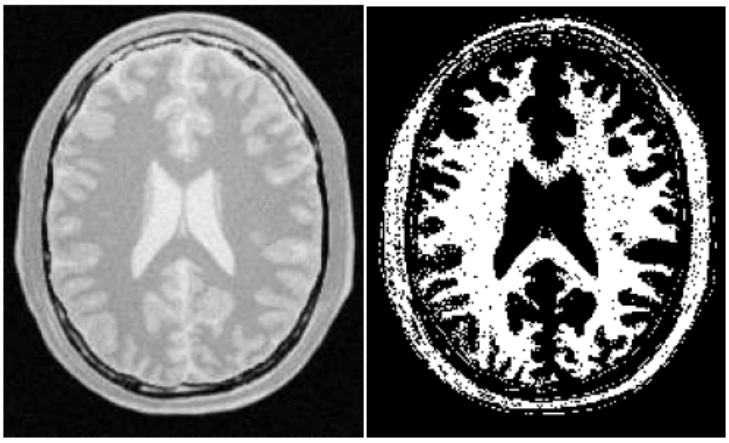


Figura 3.2.9.6.2: Resultat d'aplicar el filtre binary threshold filter a la imatge de l'esquerra

I el resultat d'aplicar aquest filtre juntament amb el *ray casting* dóna com a resultat:

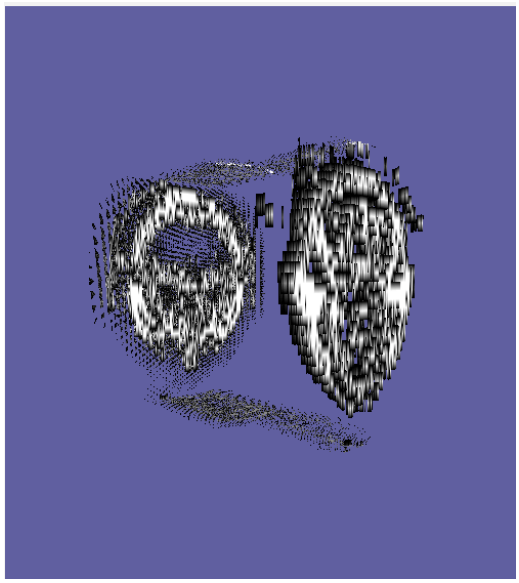


Figura 3.2.9.6.3: Resultat d'aplicar el binary threshold filter juntament amb el ray casting.

Cal dir que no és necessari descriure un interval tancat. Simplement marcant un llindar superior o inferior ja n'hi ha prou. En el cas concret de l'anterior captura, s'ha descartat tot punt amb un valor inferior a 150. És un valor força arbitrari, però fent diverses proves sembla el valor més encertat per filtrar (cal tenir en compte que al representar un punt amb un *unsigned char*, el valor mínim és el 0 i el màxim és 255). Canviant aquest valor es pot aconseguir un filtrat més flexible o menys. La idea seria incloure només els punts amb més intensitat (per això un valor tant alt).

General Threshold filter

Aquest filtre una versió un pèl diferent a l'anterior. En comptes d'assignar el mateix valor als punts que estan dins del llindar, només cal definir el valor dels de fora, l'*outsideValue*.

Llavors, si un punt està fora del llindar definit se li assigna aquest valor *outsideValue*, mentre que si està a dins se li deixa el valor que ja té.

A més, com a l'anterior, no cal definir un interval com a tal, es pot definir només un llindar superior o inferior, de manera que el valor *outsideValue* s'assigni als punts amb valor superior o inferior, respectivament, al del llindar.

Un exemple amb una imatge genèrica seria:

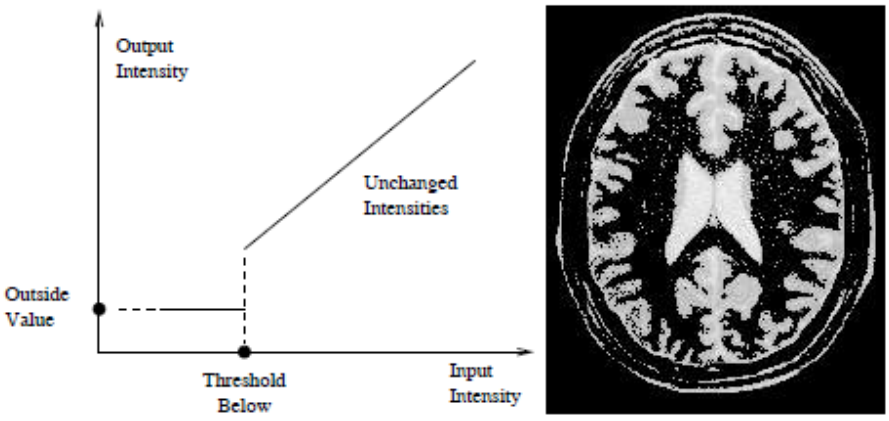


Figura 3.2.9.6.4: Filtració només per valors inferiors a un cert valor.

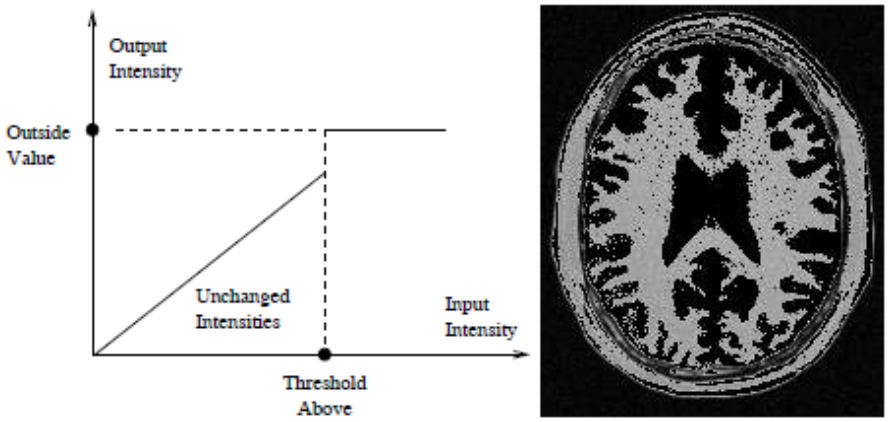


Figura 3.2.9.6.5: Filtració només per valors superiors a un cert valor.

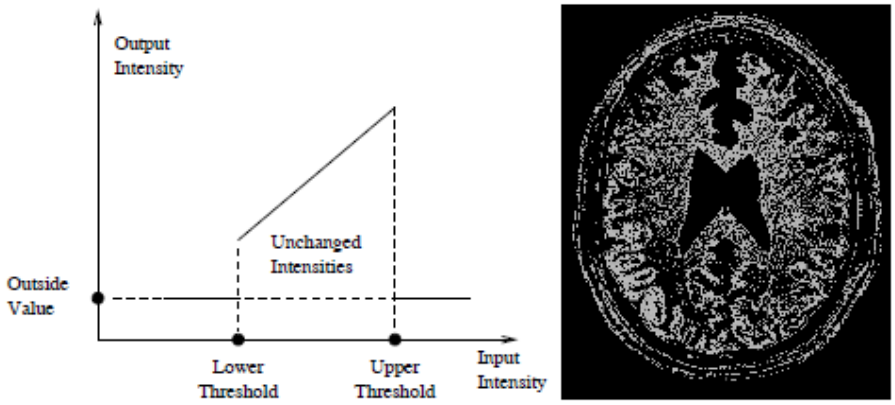
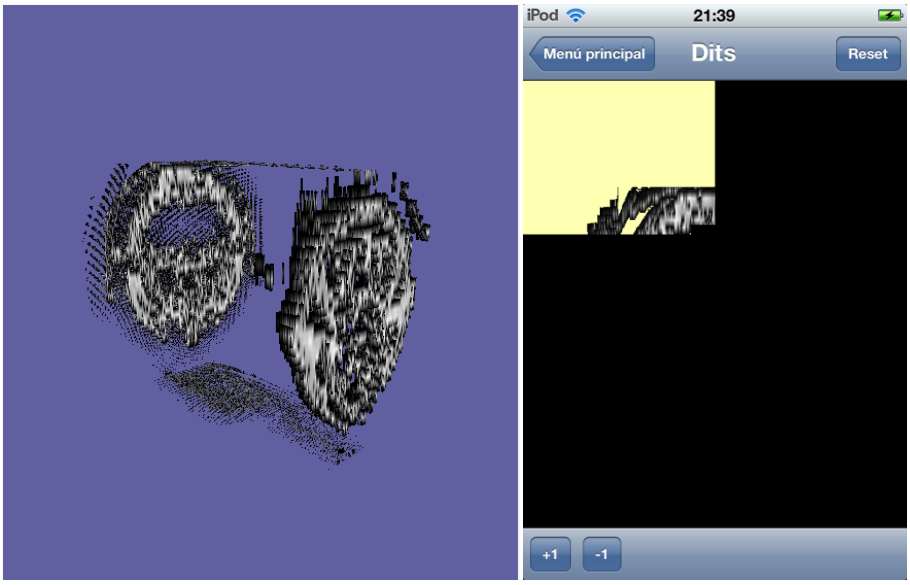


Figura 3.2.9.6.6: Filtració per valors fora d'un cert interval (amb llindar inferior i superior).

Això, aplicat a la nostra imatge amb el *ray casting*, produeix un resultat com el següent:



Figures 3.2.9.6.7 i 3.2.9.6.8: La primera és el resultat d'aplicar el ray casting juntament amb el filtre. La segona intenta fer el mateix a l'iPod però no pot.

Per fer les anteriors captures es va utilitzar també el valor 150 com a límit inferior, igual que amb el binary threshold filter. Com es pot comprovar, altre cop l'iPod es queda curt alhora de representar la imatge.

També es pot observar que, tot i aplicar la transparència necessària, no s'acaba de construir el volum com toca. Això pot ser degut a que no arriba bé la imatge al *shader* mitjançant la textura (el primer dels problemes explicats a la secció 3.2.9.5).

Amb aquests dos filtres podem solucionar el que havíem dit de que punts pròxims a negre, però no ben bé negres, no causaven transparència. Irònicament, el més simple ens serveix millor: si som nosaltres els que definim la intensitat del color mitjançant la variable *intensity* vista en l'algorisme descrit, ja ens encarreguem

nosaltres de crear la sensació de profunditat. Si deixéssim els valors originals dels punts, podríem aclarir en excés un punt amb poca intensitat lluminosa i el resultat podria ser estrany.

3.2.9.7 Optimitzacions del volume rendering

Tot i que ja no ha acabat funcionant, aquí es descriuen un seguit d'optimitzacions que estava planejat dur a terme (almenys alguna d'elles).

Two-pass raycasting a single-pass

Amb el nostre mètode, per calcular la direcció del raig s'utilitza primer el pintat del cub amb les cares posteriors i les coordenades de textura en espai 3D com a color.

Aquest pas es podria evitar. Per a fer-ho, s'hauria d'enviar a pintar el cub directament amb les cares frontals activades, i:

- Passar la posició de la càmera en *object space*[35] com a variable *uniform*.
- Calcular la distància focal entre la càmera i el cub (la utilitzada per generar la matriu de projecció).
- Al *fragment shader*, calcular una intersecció raig-cub.

El *shader* utilitzaria la posició del fragment en coordenades de dispositiu ("posició dins la pantalla") i el valor de la distància focal per calcular la direcció del raig a traçar.

Aplicant aquesta optimització aparentment senzilla, el que passaria és que s'estalviaria un pas de renderitzat, així com un accés a textura, dues coses que poden afectar suficientment al rendiment de l'aplicació com per tenir-les en compte.

Augment de la distància de mostrejat

Una de les mesures que pot accelerar dràsticament el procés en general és una de tant simple com augmentar la distància entre dues mostres en un mateix raig. Com més gran és la distància, menys mostres es poden fer en total abans d'arribar al final del volum, i menys accessos a la textura del volum cal fer (els accessos són molt cars).

El problema pot sorgir quan aquesta distància entre mostrejos s'apropa a la mida d'un voxel. El que pot passar es mostra a la següent figura:

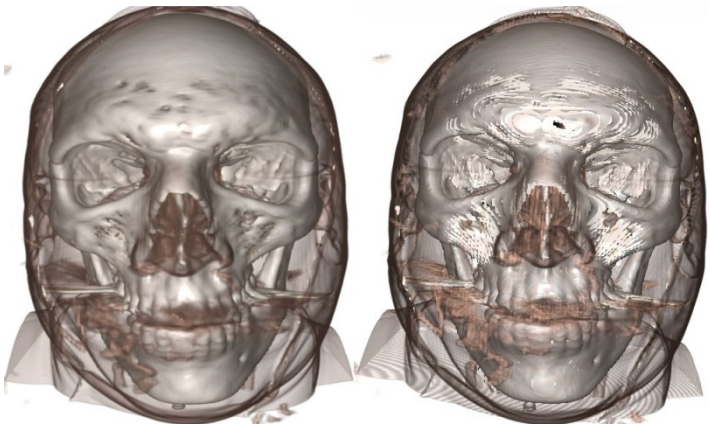


Figura 3.2.9.7.1: A la dreta, la imatge de l'esquerra representada amb una distància de mostreig 10 cops superior.

[35] Sistema de coordenades amb el que està definit cada polígon (conjunt de vèrtexs).

Tot i produir-se aquest problema, hi ha mètodes que poden suavitzar-lo. No seran descrits ja que requeririen parlar de varis conceptes dels quals no se n'ha parlat encara i no es creu necessari, sobretot tenint en compte que és una optimització que no s'ha pogut aplicar.

Interpolació de voxels

En realitat això no és una optimització en quan a rendiment, de fet l'empitjora, és una optimització del resultat, però només si s'han de fer servir textures 2D perquè les 3D no estan disponibles (com el nostre cas).

Al accedir a una textura 3D, la pròpia GPU s'encarrega d'interpolar el voxel mostrejat amb els del seu voltant, per evitar *aliasing*, però això no passa al accedir a una textura 2D. Per tant, s'hauria de fer interpolació manual. Això comportaria haver d'accedir als voxels del voltant del que intersecta amb un dels rajos, per tant, més accessos a les textures 2D.

La qualitat de la imatge milloraria, però el cost en rendiment podria ser massa elevat. Recordem que com menys accessos a les textures millor (si la textura fos 3D, la pròpia GPU té memòria cau per la textura i aquests accessos no serien tant costosos, però al ser una textura2D, no serveix).

Divisió de l'espai

Si s'utilitzés divisions de l'espai tals com arbres BSP o *octrees*, es podria reduir el càlcul de les interseccions del raig amb el volum.

El BSP, o *Binary Space Partitioning*, és un mètode que consisteix en dividir un determinat espai utilitzant plans i construint un arbre. Cada node de l'arbre conté un pla, i els seus dos fills representen el que hi ha al semiespai positiu i al semiespai negatiu del pla (a un costat o altre del pla).

Un cop dividit tant com es vol (es pot definir un nivell màxim de profunditat de l'arbre), quan es tracin rajos amb el *ray casting*, es podrà utilitzar l'arbre per determinar si es travessa una determinada regió. Això és així ja que si un determinat volum està a un subespai d'un pla, és segur que no ho estarà a l'altre. Per tant, a cada pas que s'avanci en un nivell de l'arbre es podrà descartar sempre una de les branques. Per contra, s'haurà de calcular la intersecció del raig amb varis plans (més d'un càlcul, però senzill).

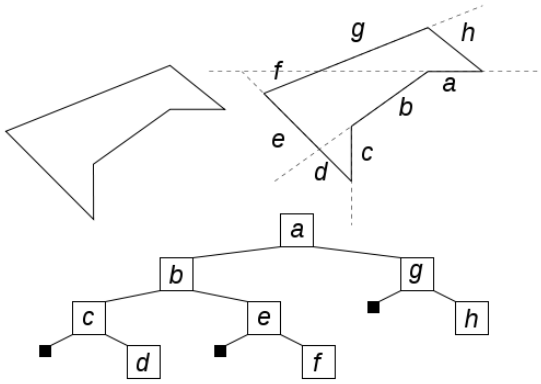


Figura 3.2.9.7.2: Divisió en arbre BSP d'un polígon.

Un *octree* és un arbre que s'aconsegueix a partir d'un cub englobant un cert volum o espai i a partir d'ell dividint en 8 cubs. Per cada cub resultant, es repeteix la operació, de manera que cada node de l'arbre té exactament 8 nodes fills, representant un d'aquests subcubs. Això es fa n vegades, segons el nivell de profunditat que es vol donar a l'arbre.

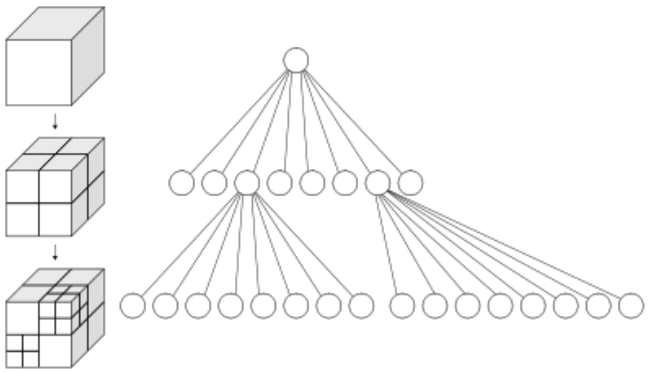


Figura 3.2.9.7.3: Formació d'un octree.

Un cop construït l'arbre, al traçar un raig primer es mira si intersecta amb el cub més gran. Si ho fa, es mira amb cada un dels fills, i així recursivament. En el moment que s'arriba a l'últim nivell (cubs més petits) i el raig segueix intersectant, llavors es mira si intersecta amb el volum "de forma estranya" que hi hagi dins aquest cub petit (les interseccions raig-cub, per contra, són força trivials). D'aquesta manera s'eviten càlculs complicats si no fan falta.

Aquest tipus d'optimitzacions no ens servirien ja que per fer la divisió es solen basar en la geometria (el volum està format per polígons) i en el nostre cas el que tenim és una imatge. La nostra geometria és un simple cub, i fer aquesta divisió no seria gens trivial. És una bona optimització pel *ray casting*, però no apte pel nostre sistema.

3.2.9.8 Altres tècniques

A part del *volume rendering*, existeixen altres tècniques que, o bé no hi ha hagut temps de provar (o estudiar si era possible), o bé eren inviables.

Per exemple, una d'elles consisteix en una variació de la que nosaltres hem definit a la secció 3.2.9.1, la de tenir varis plans.

El mètode ve a ser el mateix, la diferència seria que, per aconseguir una millor visualització de la imatge resultant quan es rotés la càmera, s'haurien de tenir 3 versions de la imatge a memòria: cada una dividida amb plans perpendiculars a cadascun dels 3 eixos. Llavors, s'escolliria quina d'aquestes versions és la més adequada per representar la imatge, depenent de l'orientació de la càmera: s'escolliria la que estigués formada pels eixos més paral·lels al vector de visió (eixos de coordenades de dispositiu).

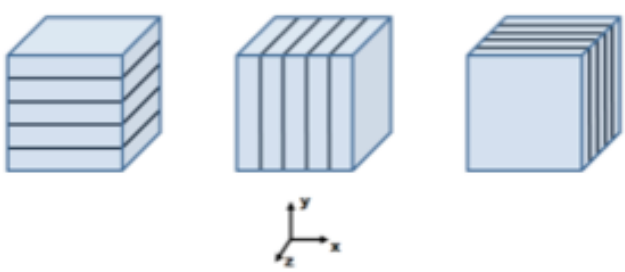


Figura 3.2.9.8.1: Les 3 maneres amb les que s'hauria de mostrejar la imatge.

El problema d'aquesta tècnica és evident: cal tenir 3 cops la imatge en memòria, cosa que resulta inadmissible.

Una altra tècnica és la proposada en *X-Ray Casting: Fast Volume Visualization Using 2D texture Mapping Techniques*. Aquesta tècnica, sense entrar massa en detall, consisteix en:

- Definir un cilindre com a geometria.
- Donada la imatge dividida en talls paral·lels al pla format pels eixos XZ, utilitzar *ray casting* des del centre de la imatge i en un nombre determinat de direccions formant una certa circumferència.
- A partir de les interseccions dels rajos amb la imatge, es calcula el color per aquells punts.
- Es guarda tota aquesta informació en una textura 2D rectangular.
    - Calculant prèviament l'equivalència de coordenades de textura del cilindre al rectangle.
- Es mapeja la textura 2D al cilindre, donant la transparència que calgui a cada punt, per formar el volum.

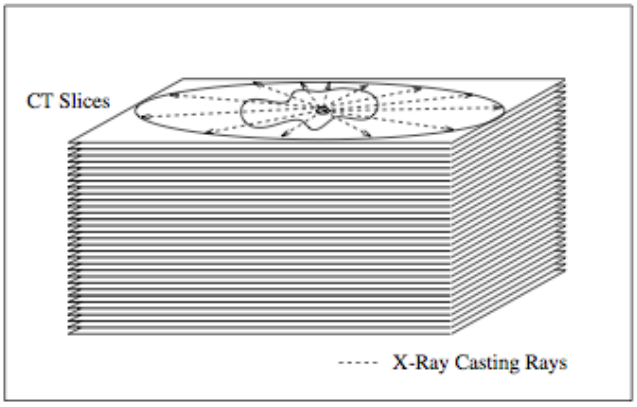


Figura 3.2.9.8.2: Pila de talls de la imatge paral·lels al pla XZ sobre els quals es fa ray casting

Una última tècnica addicional que voldria mencionar és la que es proposa a *Volume Rendering Strategies On Mobile Devices*. Aquesta tècnica, consisteix en el següent:

- Activar el *blending*.
- Definir com a geometria un seguit de plans perpendiculars a l'eix de visió.
- Definir també com a geometria un cub que englobi el volum, on la mida d'un costat equival a la mida de la diagonal del volum.
- Tenir els talls de la imatge en vàries textures 2D, tal com s'ha proposat en la solució que s'ha intentat implementar.
- Passar una matriu de transformació de les coordenades de textura, així com la típica matriu *modelview*[36] als *shaders*.
- Rotar els vèrtexs i les coordenades de textura com convingui.
- Utilitzar les coordenades de textura rotades per accedir a la textura 2D i trobar el valor del color en aquell punt.

La gràcia d'aquesta tècnica seria que els talls estan sempre perpendiculars a l'eix de visió, pel qual són estàtics i es poden guardar a la memòria cau de la GPU. Per altra banda, s'estalvia tot el *ray casting*. Però a canvi cal tenir un cub extra que englobi tot el volum i que faciliti el càlcul de les rotacions que es puguin fer, per permetre la visualització del volum des de qualsevol angle.

Tot i que no es va poder dur a terme, es va llegir una conclusió molt interessant en aquest document:

"Les GPUs de mòbil són en general menys intuïtives i més impredictibles en quan a rendiment que les GPUs d'equips de sobretaula. És molt necessari evitar condicionals en els *shaders*. Tot i que en teoria estan completament suportats, **una sola parella *if-else* pot arribar a reduir el rendiment fins a 20 frames per segon**".

Si una sola parella d'*if-else* podia afectar tant al rendiment, no era estrany que no es pogués executar el *shader* proposat, que en tenia forces (però necessaris).

[36] Matriu utilitzada en el procés de renderitzat que conté informació de la situació de la càmera dins l'escena.

3.2.10 Disseny final de l'aplicació

Independentment del fet de que l'aplicació no s'ha pogut completar amb èxit, a continuació s'adjunta l'estructura final de l'aplicació fins el punt arribat, aquest cop amb ja els atributs i operacions (almenys els més rellevants, no s'han afegit tots els atributs i totes les operacions que per defecte té un tipus de controlador).

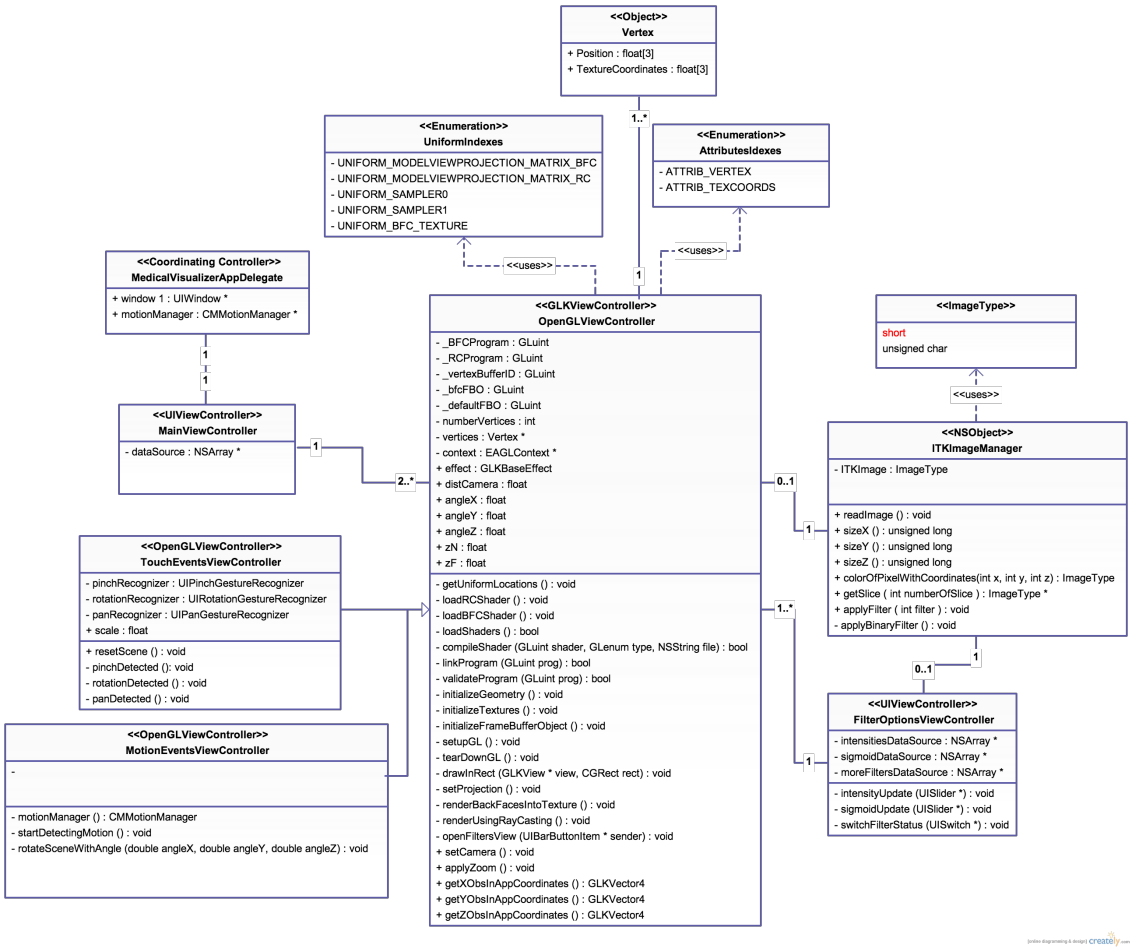


Figura 3.2.10.1: Diagrama UML de l'estructura final de l'aplicació.

Com es pot comprovar, l'estructura general no ha variat pràcticament en relació al que havíem previst en un començament, però tot i així a continuació es comentarà una mica més en profunditat cada component, ara que ja es coneixen les operacions i atributs necessaris pel funcionament de l'aplicació.

Cal dir que, per estalviar espai (i perquè no es veu necessari) no s'han afegit la majoria d'atributs i operacions que cada controlador hereta de la seva classe pare *UIViewController*, ja que farien el diagrama extremadament feixuc de llegir i, en la majoria de casos, no són rellevants pel que ens ocupa.

3.2.10.1 MedicalVisualizerAppDelegate

Tal com s'havia pensat inicialment, aquest controlador és el que coordina tota l'aplicació. Aquest controlador implementa el patró *Application Delegate*.
Aquest patró declara mètodes que són implementats per un objecte *UIApplication* singleton, en aquest cas, aquest controlador. Els mètodes mencionats són els que es llencen en events clau en l'execució de l'aplicació, com per exemple quan s'ha

acabat de carregar l'aplicació, quan l'aplicació està a punt de ser terminada o quan la memòria s'està acabant. Implementar aquests mètodes permet respondre adequadament a aquests events.

Per falta de temps, no s'hi ha dedicat l'especial atenció que es requeria per poder alliberar correctament els recursos en el moment que l'aplicació es pausa (perquè l'usuari prem el botó "*home*" o perquè s'apaga el dispositiu). El problema de la memòria venia donat per la mateixa execució de l'aplicació, però si s'hagués arribat a solucionar, s'hauria d'haver implementat els mètodes d'aquest protocol per assegurar-se que múltiples execucions de l'aplicació (sortint, tornant a entrar, etc.) no generaven un augment en l'ús de la memòria, és a dir, que els recursos eren pròpiament alliberats

De totes maneres, l'*automatic reference counting* (ARC) introduïda amb la versió 5.0 d'iOS, facilitia moltíssim la gestió de la memòria. En versions anteriors, calia alliberar els recursos manualment, mentre que ara gran part es fa automàticament.

El que sí es va implementar, que respon al protocol *Application Delegate*, és l'adquisició de l'objecte *CMMotionManager*, l'objecte que proporciona la informació que capten els sensors com l'acceleròmetre o el giroscopi. Es va implementar un mètode per obtenir una única instància d'aquest objecte, per pròpia recomanació d'Apple. El motiu que insta a tenir només una instància d'aquest objecte és que si se'n té més d'una l'interval amb el que es reben dades actualitzades pot variar de forma inconsistent. Per això, com és un mètode que ha de ser únic per tota l'aplicació, pertany a aquest controlador.

3.2.10.2 MainViewController

Com s'havia previst, aquest controlador és del tipus *navigation controller*. Per donar les diferents opcions de navegació (utilitzar la visualització amb gestos tàctils o amb el giroscopi), conté una taula i implementa els protocols *UITableViewDataSource* i *UITableViewDelegate*.

El primer protocol marca l'origen de les dades que serveixen per omplir la taula.
El segon, indica què cal fer quan l'usuari selecciona una de les caselles. En aquest cas, s'aplicaria la navegació carregant el controlador al que pertany la vista d'OpenGL on en teoria es mostra la imatge.

En el diagrama UML anterior, la propietat *dataSource* és la que conté les opcions seleccionables.

3.2.10.3 OpenGLViewController

Aquest controlador és el que ha absorbit la gran majoria del temps dedicat a aquest projecte. És el que conté la base de la representació de l'escena 3D mitjançant OpenGL ES.

Entre les seves responsabilitats està l'inicialització dels *shaders*, dels objectes necessaris per a la representació (*framebuffer object*, *vertex array object*, etc.) i el bucle de renderitzat principal. En aquest, es renderitza primer les cares posteriors del cub que es té com a geometria i posteriorment es renderitza (o s'intenta) la imatge mèdica, utilitzant el *volume rendering*.

En la vista d'aquest controlador, s'utilitza la *navigation toolbar* proporcionada pel *navigation controller* per mostrar un botó "Filtering options" que el que fa és carregar el controlador corresponent amb la vista de les opcions de filtres.

També consta d'uns mètodes per obtenir els eixos de coordenades de l'observador en el sistema de coordenades de l'escena, *getXObsInAppCoordinates*, *getYObsInAppCoordinates* i *getZObsInAppCoordinates*, necessaris per aplicar les rotacions sobre l'escena.

Finalment, aplica el zoom i les rotacions utilitzant els mètodes *applyZoom* i *setCamera* respectivament, quan és necessari.

3.2.10.4 TouchEventsViewController

Si en la vista carregada pel *MainViewController* es selecciona l'opció d'utilitzar els gestos tàctils, es carregaria aquest controlador, que és una extensió de l'anterior.

Aquest controlador implementa el protocol *UIGestureRecognizerDelegate*, que vol dir que aquest controlador serà responsable de tractar events de gestos sobre la pantalla tàctil.

Concretament, detecta els gestos de tipus "pessic", rotació i pan i els utilitza per rotar l'escena i fer zoom. Els atributs que té aquest controlador són els que serveixen per detectar aquests gestos. Més detalls a la secció 4.2.

Addicionalment, es va veure necessari implementar un mètode per reiniciar l'escena a l'estat inicial, ja que, tot i que és possible tornar-la-hi mitjançant els gestos inversos, podia resultar feixuc. Per a fer-ho es va afegir un botó "Reset" a la *navigation bar*, que bàsicament reinicia l'estat de la càmera de l'escena a la posició inicial.

3.2.10.5 MotionEventsViewController

Aquest controlador hereda del controlador *OpenGLViewController*, i és el que es carrega quan des de la vista carregada pel *MainViewController* es selecciona l'ús dels moviments del dispositiu per visualitzar l'escena.

Aquest controlador obté l'instància de *CMMotionManager* que inicialitza el controlador *MedicalVisualizerAppDelegate* i li afegeix una cua a la que escoltar els events provocats pel moviment del dispositiu, així com una funció *handler* que respongui a aquests events. Utilitzant aquests events, es rota la càmera de l'escena representada. Més detalls a la secció 4.3.

3.2.10.6 ITKImageManager

Aquesta classe és l'encarregada d'interaccionar amb la llibreria ITK. S'encarrega de llegir la imatge del fitxer i d'obtenir informació de la imatge necessària per pintar-la. Un dels mètodes clau és el *getSlice*, que com s'havia comentat anteriorment, servia per obtenir un punter a la posició de memòria on hi havia un determinat tall de la imatge ITK.

A més, també implementa un mètode per l'aplicació de filtres i el mètode per aplicar el *Binary Threshold Filter*. Si hi hagués hagut més temps, i s'hagués aconseguit representar la imatge en 3D, aquest primer mètode s'hagués vinculat amb el controlador que s'explica tot just a continuació, de manera que es rebessin peticions d'aplicar un filtre o modificar algun dels paràmetres d'algun.

En el diagrama anterior, *short* està en vermell perquè va ser el tipus d'imatge que es va haver de descartar, i que per això es va acabar utilitzant el tipus *unsigned char*.

3.2.10.7 FilterOptionsViewController

Aquest controlador ha quedat com una primera implementació del que seria l'encarregat d'ordenar l'aplicació de filtres. A partir de la vista de la figura 5.5.1, que es veurà més endavant, s'obtindrien events en el canvi dels valors d'algun dels widgets, de manera que es notifiqués la classe *ITKImageManager* i aquesta apliqués els canvis corresponents.

Com es veu a la figura 5.5.1, aquest controlador també conté una taula amb el que implementa els protocols *UITableViewDataSource* i *UITableViewDelegate*, com el *MainViewController*.

4. Visualització de l'escena

En aquesta secció es comenta el que es va fer per implementar les tècniques de visualització que ajuden a percebre correctament una escena 3D: el zoom, el pan i les rotacions.

Abans de res, cal dir que aquesta secció, si hem dit que aquests apartats de la memòria estaven ordenats de forma temporal, no pertocaria que estigués aquí. Seguint l'ordre temporal estrictament, aquesta secció hauria d'anar després de que quedés constatat que no era possible utilitzar les funcions *glBegin* i *glEnd* per definir la geometria necessària. Com que s'havia d'utilitzar un sistema fins llavors desconegut per a pintar geometria (encara no s'havia tocat res a l'assignatura de gràfics avançada), es va concentrar els esforços en representar un simple cub de colors i llavors provar d'aplicar les diferents tècniques de visualització.

El motiu pel qual s'ha deixat la descripció d'aquesta secció per més endavant, doncs, és que d'aquesta manera s'ha pogut agrupar molt més fàcilment l'explicació de la part de la representació en 3D de la imatge, que era el "plat principal" del projecte.

4.1 Visualitzció amb ITK.

Un apunt que cal fer és que ITK permet aplicar transformacions visuals com les que es volien aplicar. Permet escalar, rotar i fins i tot traslladar. El problema és que només es permet fer en dues dimensions, equivalents al pla de visió.

Si no fos per aquest inconvenient, la imatge s'hagués pogut representar utilitzant un cert nombre de plans, semblant a la tècnica descrita a la secció 3.2.9.1. Llavors, es podria utilitzar els events dels gestos aplicats sobre la pantalla tàctil, per exemple, per transformar la imatge ITK, sense haver d'afectar a la geometria d'OpenGL.

Com que no pot fer-se així, es va haver de recórrer a les transformacions pròpies d'OpenGL.

4.2 Visualització utilitzant gestos tàctils

Tot i que es va plantejar la possibilitat d'utilitzar el giroscopi per a aplicar rotacions a l'escena, el primer que calia fer era realitzar aquests moviments a partir de gestos tàctils aplicats a sobre la pantalla del dispositiu, els anomenats gestos *multitouch*[37].

Per a fer-ho, es van fer servir els anomenats *gesture recognizers* ("reconeixedors de gestos"). Aquests són uns objectes que s'afegeixen a la vista i que permeten respondre a accions d'igual forma que ho faria un controlador. Els propis reconeixedors interpreten els gestos tàctils per determinar de quin tipus de gest es tracta (un sol toc a la pantalla, un toc continu, un toc més un desplaçament del dit, etc.). Si el gest és correctament reconegut, el reconeixedor corresponent envia un

[37] S'anomenen *multitouch* perquè poden reconèixer un o més dits (gestos) simultàniament.

missatge a l'objecte al qual està lligat, que típicament sol ser el controlador de la vista a la qual s'ha reconegut el gest.

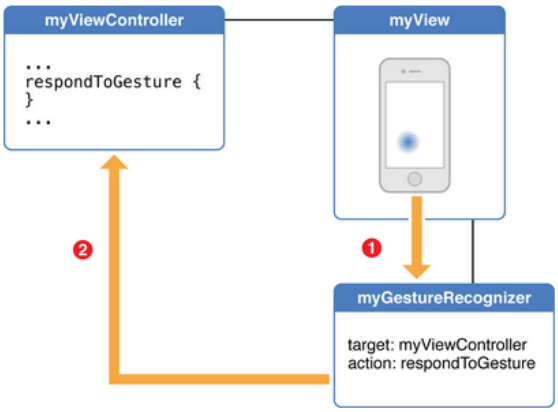


Figura 4.2.1: Estructura que segueixen els events provocats per gestos.

A l'anterior figura es pot veure com el dispositiu envia l'event del gest reconegut al reconeixedor i aquest fa saltar l'acció corresponent del controlador al que està lligat.

Tot i que es poden definir reconeixedors de gestos personalitzats, per defecte n'hi ha uns de ja definits en el framework UIKit[38]. Alguns d'aquests, i que vam acabar utilitzant, són:

- Detector de rotacions, *UIRotationGestureRecognizer*.
- Detector de moviments de tipus pan, *UIPanGestureRecognizer*.
- Detector de moviments de tipus pessic, *UIPinchGestureRecognizer*.

Per aclarir més quin tipus de moviments són els que reconeixen, a continuació s'exposen figures on es pot veure millor.

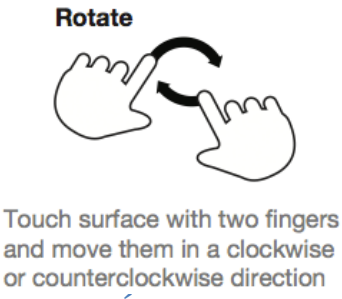


Figura 4.2.2: Rotació. És el moviment reconegut per UIRotationGestureRecognizer.

[38] Prové les classes necessàries per construir i manejar la interfície gràfica d'un aplicació per iOS. Conté, per exemple, gestió d'events, un model de pintat, finestres, vistes i controladors dissenyats especialment per a la interfície de les pantalles tàctils.

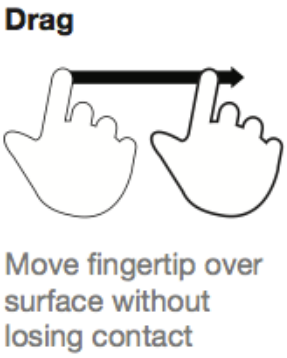


Figura 4.2.3: Arrossegament. Moviment detectat per UIPanGestureRecognizer.

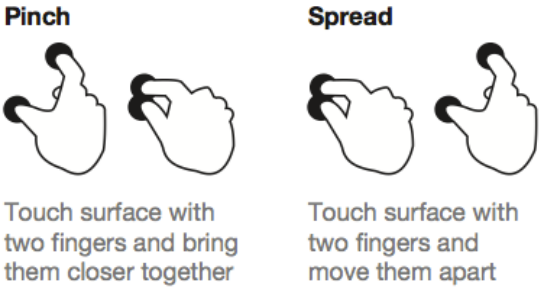


Figura 4.2.4: Pessic (cap endins i cap a fora). Moviment detectat per UIPinchGestureRecognizer.

El moviment de rotació i d'arrossegament es van fer servir per rotar el cub (el primer rotaria la geometria segons el moviment descrit pels dits, i el segon la rotaria segons la direcció marcada pels dits).

El moviment del pessic es va fer servir pel zoom (apropar/allunyar la geometria).

Faltava doncs, un moviment pel pan. Tot i que, quan es buscava com es podia fer, es va arribar a la conclusió que era un moviment força prescindible, més aviat opcional, ja que amb la combinació dels dos moviments anteriors es podia veure qualsevol punt de la geometria representada. No oblidem, tampoc, que es van implementar aquest tipus de moviments abans que intentar representar la imatge en 3D precisament per permetre interaccionar amb la imatge creada, i poder veure la imatge des de diferents angles, per poder comprovar si es formava bé o no.

No només això, sinó que amb els reconeixedors que existien per defecte no eren suficient per poder detectar un moviment que pogués ser utilitzat pel pan. Per tant, caldria crear un reconeixedor personalitzat, i això era una tasca que calia prorrogar fins un cop tota la resta funcionés, punt al que per desgràcia no s'ha arribat.

4.2.1 Zoom

Per implementar el zoom el que es fa és detectar si el gest de pessigar es fa apropant els dits (en la figura anterior, *pinch*) o si pel contrari es separen els dits (en l'anterior figura, *spread*).

Però, per defecte, això no es pot distingir. El reconeixedor no pot saber-ho per si sol.

Tot i així, el *UIPinchGestureRecognizer* té una propietat *scale* (escala). Aquesta propietat indica la distància entre els dos punts tocats a la pantalla, en coordenades de dispositiu.

Per tant, el que es fa és:

- Cada cop que es detecta un event d'aquest tipus s'emmagatzema el valor d'aquesta propietat.
- Es compara el valor ja emmagatzemat amb el nou valor.
    - Si el nou valor és inferior, el moviment és de *pinch* (els dits s'ajunten) → cal allunyar la imatge.
    - Si el nou valor és superior, el moviment és de *spread* (els dits s'allunyen) → cal apropar la imatge.

Un cop obtinguda la direcció en que es fa és apropar o allunyar, segons convingui, la imatge. S'utilitza un valor constant per a determinar quina distància apropar-se o allunyar-se.

Per aplicar-lo, només cal aplicar una translació a la matriu que emmagatzema totes les transformacions a fer els vèrtexs per representar l'escena. La matriu per fer aquesta translació és:

$$M_T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad d = distància\ a\ apropar/allunyar$$

El que cal fer és que la matriu de visualització esdevingui el resultat de multiplicar l'actual matriu de visualització per aquesta matriu de translació. Llavors, aquesta matriu M_T serà la primera a multiplicar-se amb els vèrtexs de la geometria a representar. Això farà que s'allunyin (o apropin, depenent del signe de *d*) una certa distància *d* del que vindria a ser el punt on es troba la càmera de l'escena, en la direcció de l'eix Z de l'observador (perpendicular a la pantalla del dispositiu).

Amb el procés descrit, es va aconseguir implementar el zoom.

4.2.2 Rotacions

Per simular les rotacions calen dos gestos diferents: el pan (irònicament, en el nostre cas, ens serveix per rotar, no per fer pan) i el de rotació amb dos dits movent-se en sentit horari o antihorari.

Aquest cop sí que es pot detectar la direcció del gest. La classe *UIRotationGestureRecognizer* disposa de la propietat *velocity*, que indica la velocitat de la rotació en radis per segon. La classe *UIPanGestureRecognizer* disposa de la propietat *velocityInView*, que indica el vector velocitat, en punts per segon, amb que es produeix el pan.

Així doncs, per detectar la direcció es pot utilitzar el signe d'aquestes propietats:

- En el cas de *velocity*, si és positiva és que la rotació es produeix en sentit horari.
- En el cas de *velocityInView*:
    - La component x serà positiva si el gest produeix un desplaçament d'esquerra a dreta. D'altra banda serà negativa.
    - La component y serà positiva si el gest produeix un desplaçament de dalt cap a baix. D'altra banda serà negativa.

Per aplicar les rotacions el que cal també conèixer són els eixos de coordenades de l'observador: un moviment totalment horitzontal sobre la pantalla tàctil ha de moure la figura segons aquest eix, no segons l'eix local de l'escena.

Aquests eixos d'observador es poden extreure fàcilment de la matriu de visualització en qualsevol moment:

$$M_V = \begin{pmatrix} X_{XObs} & Y_{XObs} & Z_{XObs} & W_{XObs} \\ X_{YObs} & Y_{YObs} & Z_{YObs} & W_{YObs} \\ X_{ZObs} & Y_{ZObs} & Z_{ZObs} & W_{ZObs} \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

La primera fila correspon a l'eix X de l'observador en coordenades de l'escena (de món), la segona a l'eix Y i la tercera a l'eix Z.

Així doncs, per fer les transformacions corresponents, només cal aplicar rotacions sobre aquests eixos.

La matriu necessària per aplicar aquesta transformació no és trivial, però per sort existeix una funció de GLKit[39] que és *GLKMatrix4RotateWithVector4*. A aquesta funció se li passa una matriu, un angle i un vector i el que fa és aplicar una rotació sobre la matriu ja existent en l'eix que marca el vector i de tant com marqui l'angle.

Cal tenir en ment que els eixos X i Y estan invertits, és a dir, quan fem un gest horitzontal, tot i que el que varia és la velocitat en X, el que hem de fer és rotar respecte l'eix Y visual, de manera que la figura representada roti "rodant" sobre l'eix Y. I al revés igual, moviments verticals equivalen a rotacions sobre l'eix X del dispositiu.

Aplicant l'explicat les rotacions van quedar implementades.

4.3 Visualització utilitzant el giroscopi

Com que s'havia aconseguit relativament fàcilment el moure el cub utilitzant els gestos sobre la pantalla tàctil, el que es va fer a continuació va ser intentar aplicar el mateix tipus de moviments utilitzant el giroscopi del dispositiu.

[39] Llibreria amb un conjunt de rutines matemàtiques.

Primer de tot, crec necessari distingir entre giroscopi i acceleròmetre, ja que són comunament confosos i hom podria preguntar-se si no era l'acceleròmetre el que servia per a rotar la figura utilitzant la inclinació del dispositiu:

Un **acceleròmetre** té la capacitat per mesurar l'orientació d'una plataforma fixa respecte a la superfície terrestre. Mesura l'equivalent a tenir una massa situada al dispositiu i sobre la que actua la força de la gravetat. Per tant, si la plataforma es trobés en caiguda lliure, l'acceleració mostrada seria 0, ja que es troba en un context on la massa no té pes.

Per altra banda, el **giroscopi** té la capacitat de mesurar la rotació sobre un determinat eix. Mesura l'orientació a l'espai d'un determinat dispositiu.

Per explicar la diferència de forma senzilla: si agafem un dispositiu sense giroscopi, i el mantenim just davant nostre mirant cap al front i donem mitja volta sobre nosaltres mateixos, el dispositiu no notarà cap canvi, ja que, per a ell, estarà a la mateixa posició que abans.

En canvi, si fem el mateix en un dispositiu amb giroscopi, el dispositiu percebrà aquest canvi quan rotem.

Perquè necessitàvem utilitzar el giroscopi i no l'acceleròmetre, doncs? Perquè el que volíem és conèixer en tot moment l'orientació del dispositiu, independentment de la velocitat i temps amb la que s'hagi fet la rotació.

Si utilitzéssim l'acceleròmetre, deixant el dispositiu inclinat, aniríem rebent constantment events d'acceleració, cosa que no volem: podria provocar que anéssim rotant infinitament la figura segons la inclinació del dispositiu, ja que no tindríem manera de distingir quan el dispositiu s'ha quedat en una posició fixe.

En canvi, amb el giroscopi, al deixar el dispositiu inclinat, rebem informació de la seva orientació i podem obrar en conseqüència. Som capaços de distingir quan el dispositiu està parat.

4.3.1 Rotacions

A diferència dels moviments causats per gestos aplicats sobre la pantalla tàctil, utilitzant el giroscopi només podíem aplicar rotacions, ni zoom ni pan.

Es podria incloure el zoom utilitzant els gestos tàctils, com a l'anterior secció, però resulta difícil aplicar el zoom depenent de l'orientació a la que es trobi el dispositiu, així que es va decidir no barrejar els dos tipus d'events diferents i només es va permetre utilitzar o gestos o giroscopi, de forma excloent.

Per rotar la geometria utilitzant el giroscopi el que es fa és editar la posició de la càmera de l'escena, rotant-la segons convingui.

En OpenGL, per a definir aquesta posició de la càmera típicament es fa el següent:

1. Definir una matriu identitat.
2. Rotar segons l'angle de l'eix X.

3. Rotar segons l'angle de l'eix Y.
4. Rotar segons l'angle de l'eix Z.
5. Allunyar una certa distància *d* la càmera del centre de l'escena, per poder visualitzar l'escena (d'altra banda, la càmera es trobaria en aquest centre de l'escena, i no es podria contemplar tota sencera).

Aquesta matriu és la primera de les matrius que es multipliquen amb els vèrtexs de la geometria a representar, per situar ja la càmera a la posició final.

Així doncs, per aplicar les rotacions utilitzant el giroscopi el que cal fer és situar la càmera de l'escena d'acord amb els angles que s'obtinguin del giroscopi.

El giroscopi proporciona vària informació i en varis formats més o menys redundants, però unes de les propietats que són consultables en tot moment són els tres angles que forma el dispositiu, segons els tres eixos, respecte a la posició inicial.

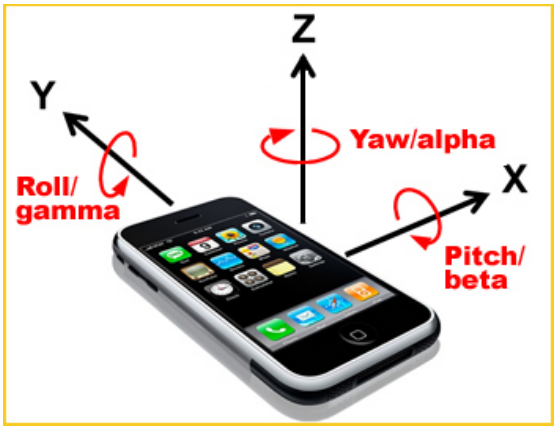


Figura 4.3.1: Mostra dels valors recollits pel giroscopi.

Tenint en compte que l'estat inicial és amb el dispositiu pla (recolzat sobre una taula, per exemple):

- El valor de *pitch* representa rotacions segons l'eix X.
- El valor de *roll* representa rotacions segons l'eix Y.
- El valor de *yaw* representa rotacions segons l'eix Z.

Per tant, per tal d'aconseguir produir rotacions amb el giroscopi el que es va fer va ser actualitzar els angles sobre els eixos X, Y i Z amb els valors *pitch*, *roll* i *yaw*, respectivament, proporcionats per ell. Això actualitzava la matriu de la càmera i per tant, l'escena.

5. Opcions de filtres útils

Aquesta secció serveix per explicar una mica els filtres que s'haguessin pogut intentar aplicar si la imatge s'hagués representat correctament en tres dimensions. No vol dir que siguin els únics, però sí els que, quan es va investigar el manual de la llibreria ITK, se'ls veia amb més potencial.

Abans, però, de parlar de cap filtre en concret, cal recordar el que es va descobrir en el seu moment:

Per tal evitar perdre la imatge original, cada filtre que s'aplica ho fa creant-ne una còpia en un nou espai de memòria. El filtre reserva la memòria que necessita per a representar la nova imatge i llavors aplica les transformacions a la imatge original i les guarda en la nova memòria reservada.

La quantitat d'aquesta memòria reservada no té perquè ser igual a la que necessita la imatge abans d'aplicar el filtre, ja que si es canvia el tipus de dada que representa un punt, aquesta quantitat pot augmentar o disminuir.

Per tant, en un context on la memòria va més aviat justa, i en el que recordem que el sol fet de llegir la imatge amb el seu tipus de dada original, l'*unsigned short*, ja ens feia rebre un *memory warning* (i fins i tot havia provocat que es tanqués l'aplicació), res no assegura al 100% que els filtres que es descriuen a continuació poguessin ser aplicables sense que l'aplicació es tanqués per falta de memòria.

El que tampoc es podia fer és perdre la imatge al aplicar un filtre, amb el que sempre caldria mantenir la imatge original a memòria, ocupant les 9 MB de memòria només per estar disponible.

5.1 Binary Threshold Filter

D'aquest filtre ja n'havíem parlat amb anterioritat. Servia per definir un llindar, o un interval, i assignar un mateix valor a tots els punts de dins i el mateix valor a tots els punts de fora.

Tot i que per a representar la imatge es va fer servir uns valors concrets, es podria donar l'opció a modificar dinàmicament aquests valors utilitzant, per exemple, dos *sliders*, per tal d'escollir aquests valors que limiten l'interval.

Figura 5.1.1: Slider d'iOS

Al permetre modificar aquests valors dinàmicament, el que s'estaria fent és definir quin interval de punts volem visibles i quins no. D'aquesta manera, podríem discriminar els punts de la imatge segons intensitat. Això podria servir, en una imatge mèdica, per exemple per descartar punts de la pell (valors més febles d'intensitat lluminosa) i només representar part de l'esquelet (valors més forts

d'intensitat lluminosa) o agafar només valors d'intensitat mitja, que podrien representar massa muscular, etc.
Amb les combinacions apropiades de valors, podria veure's qualsevol part de la imatge, i realment discriminar esquelet en front de la resta, per exemple, pot ser extremadament útil.

Aquest era el primer filtre que es volia aplicar tant bon punt la imatge hagués estat representada en 3D. Era simple d'introduir i alhora tenia un gran potencial.

5.2 Sigmoid Image Filter

El *Sigmoid Image Filter* és un filtre que el que fa és transformar un rang d'intensitats a un nou rang, fent una transició contínua i molt suau en els límits del rang.

Un sigmoide és una funció que "té forma d'essa". Es fa servir per concentrar l'atenció en un particular conjunt de valors i progressivament atenuar els valors fora d'aquest rang.

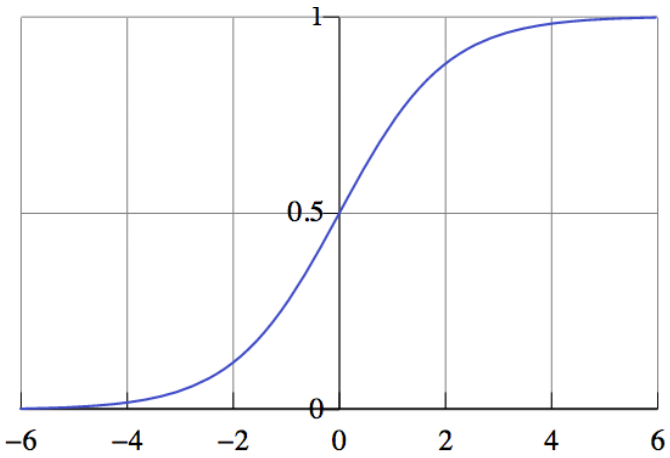


Figura 5.2.1: Exemple de funció sigmoide.

La llibreria ITK proporciona 4 paràmetres diferents per tal de condicionar la corba que fa aquesta funció.

L'equació que proporciona ITK és:

$$I' = (Max - Min) \cdot \frac{1}{\left(1 + e^{-\left(\frac{I-\beta}{\alpha}\right)}\right)} + Min$$

En aquesta equació:

- La *I* representa la intensitat del píxel tractat.
- La *I'* representa la intensitat final calculada.
- *Min* i *Max* són els valors mínim i màxim de la imatge resultant.
- L'alfa defineix l'amplada del rang d'intensitats.
- La beta defineix el valor de la intensitat sobre la qual es centrarà el rang.

En la següent figura es mostra com variaria la corba de la funció segons diferents paràmetres:

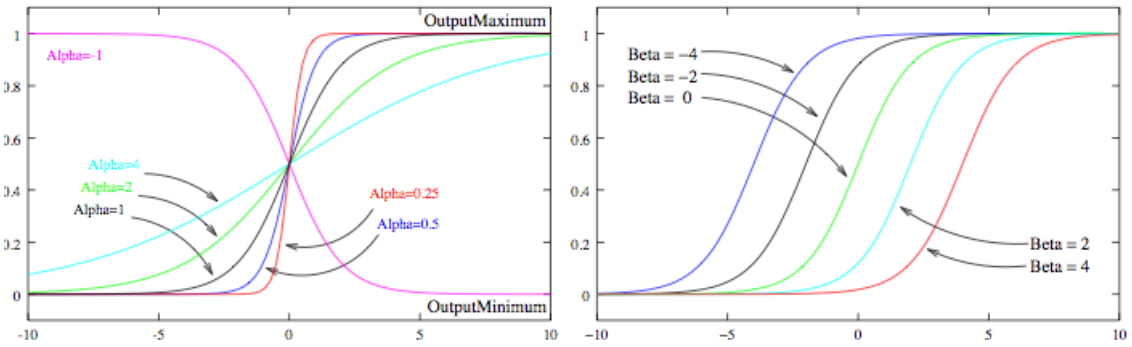


Figura 5.2.2: Corbes dibuixades variant els paràmetres de l'anterior equació.

Un exemple de com quedaria l'aplicació d'aquest filtre és:

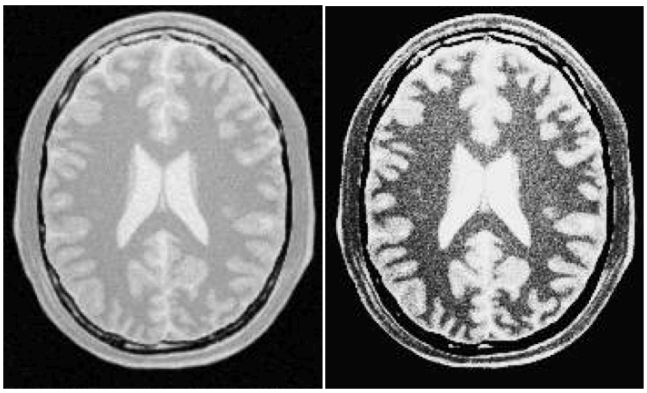


Figura 5.2.3: Efecte del filtre Sigmoid.

Aquest filtre es podria utilitzar conjuntament amb l'anterior. Primer es transformaria la imatge utilitzant el *Segmoid Image Filter*, de manera que un donat rang d'intensitats estigués més ressaltat. Després, amb el *Binary Threshold Filter*, deixaríem de banda les intensitats que haguessin resultat més febles per representar només aquelles que són més fortes. Això aconseguiria una representació de la imatge més precisa, i podria ajudar a evitar soroll[40].

[40] Punts que, per errors electrònics o de precisió del sensor que ha captat la imatge, tenen valors que no corresponen, i per tant són representats per una intensitat lluminosa que no toca.

El problema que presenta aquest filtre és com muntar la interfície gràfica, per tal de facilitar l'ús a l'usuari. Hi ha forces paràmetres a tenir en compte per fer els càlculs.

Ara bé, probablement afegint dos *sliders* més als proposats a l'anterior apartat ja en faríem:

El primer *slider* seria pel valor de la beta, en l'anterior equació. El segon seria pel valor d'alfa.

Llavors, es podria utilitzar el valor dels altres dos *sliders* per als valors *Min* i *Max*.

El valor de cadascun dels extrems de l'interval sobre el qual aplicar el *Binary Threshold Filter* podrien ser constants o calculats interiorment, sense interacció amb l'usuari.

5.3 Median Image Filter

Aquest filtre s'utilitza com a aproximació robusta per la reducció del soroll. És particularment efectiu contra el soroll *salt-and-pepper*[41].

Figura 5.3.1: Imatge amb soroll tipus "sal i pebre".

Per reduir aquest tipus de soroll, aquest tipus de filtre calcula el valor de cada punt de sortida com la mitjana estadística dels valors dels píxels al voltant del píxel tractat.

La quantitat de punts veïns a consultar és un paràmetre que pot ser introduït per l'usuari. La següent figura mostra l'efecte d'aquest filtre aplicat a un conjunt de valors representant una imatge 2D:

[41] El tipus de soroll de "sal i pebre" és el tipus de soroll al que s'anomenen els punts blancs i negres aleatòriament distribuïts sobre una imatge.

28	26	50
27	25	29
25	30	32

→ 28

Figura 5.3.2: El valor de la dreta és el resultat d'aplicar el filtre al valor del centre de la graella de l'esquerra.

El resultat d'aplicar aquest filtre a una imatge mèdica es pot veure en la següent figura:

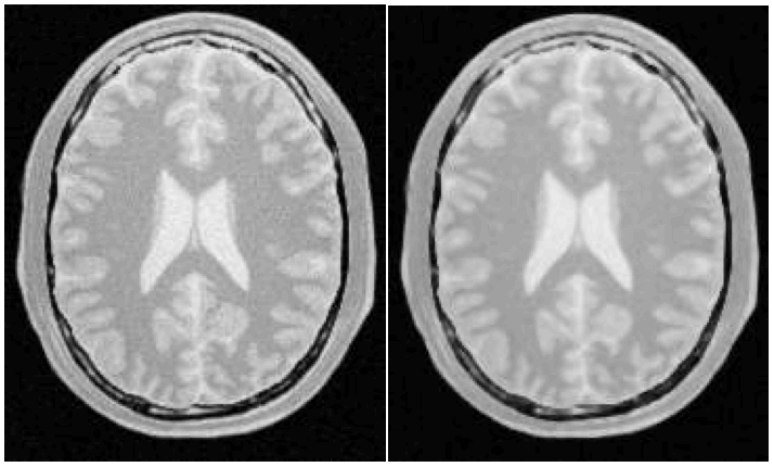


Figura 5.3.3: A la dreta el resultat d'aplicar el filtre Median Image Filter a la imatge de l'esquerra.

Tot i que costi una mica de percebre, per la mida de la imatge, es pot veure com les zones amb un mínim de soroll apreciable són més homogeneïtzades.

No es té molt clar si aquest filtre es podria aplicar a la imatge en 3D, caldria fer la prova, però si realment disminuís el soroll de la imatge, seria interessant. Ara bé, al homogeneïtzar més la imatge, es perd definició en el que vindria a ser les fronteres de les diferents seccions distingibles de la imatge. Per això mateix, podria ser un filtre que en determinats moments fóra útil i en determinats moments fos més un problema.

Per això mateix, una proposta seria permetre activar o desactivar el filtre utilitzant un *switch* per permetre activar o desactivar el filtre.

Figura 5.3.4: Switch d'iOS

5.4 Filtre personalitzat

Una altra opció hagués estat crear un filtre personalitzat. Tot i que, com s'ha dit a la secció 3.2.3.2, no era gens fàcil, si s'hagués aconseguit representar la imatge hagués estat viable intentar-ho.

Un filtre que hagués sigut interessant implementar seria un que transformés cada punt de la imatge a una estructura que contingués la intensitat de luminància, igual que l'original, però a més un valor més, guardat en un *unsigned char*.

Aquest valor es calcularia a partir del valor de la luminància, i el que faria seria indexar la posició d'una array definida al *fragment shader* on hi tindríem guardats uns colors base.

La idea seria marcar uns certs rangs d'intensitats i que cada un correspongués a un d'aquests colors base. Llavors, en comptes de representar la imatge en escala de grisos, s'utilitzaria la intensitat lluminosa de la imatge en aquell punt, aplicada sobre el color base, per representar aquell punt.

La intenció de tot plegat seria intentar d'alguna manera pintar diferent la pell, els músculs o els ossos de la imatge. Probablement fos difícil distingir el rang d'intensitats de cada cosa, però caldria provar-ho per saber-ho del cert.

Costa trobar una imatge definint el que s'està descrivint, ja que aquest filtre es té en ment, però no s'ha realitzat. A continuació s'adjunta una imatge del que més s'assembla al que s'intenta descriure:

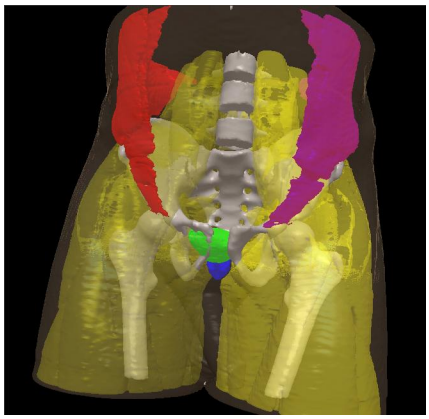


Figura 5.4.1: Representació semblant al que seria l'aplicació del filtre: diferents parts tenen un color base diferent.

Pot semblar que aquest filtre és innecessari, que es podria comprovar el valor de la intensitat des del *fragment shader* i accedir a la array igualment, o fins i tot utilitzar una textura d'una dimensió. Però des de que es va descobrir com de dolents eren els condicionals en *shaders*, semblava que el fet d'ocupar un byte més per punt no era tant dolent com fer un condicional més (tot i que això faria saltar el *memory warning*). Accedir a l'array utilitzant aquest byte era molt menys costós.

També és cert que aquest filtre no esborraria els valors originals llegits de la imatge, amb el que seria perfectament viable tenir-lo sempre actiu i simplement donar l'opció de fer servir l'array de colors o no, utilitzant per exemple un altre *switch*.

5.5 Vista d'opcions dels filtres

Segons els filtres plantejats, es va dissenyar la vista que podia ser accedida des de qualsevol de les dues pantalles de visualització de la imatge: la que utilitza els gestos tàctils o la que utilitza el giroscopi.

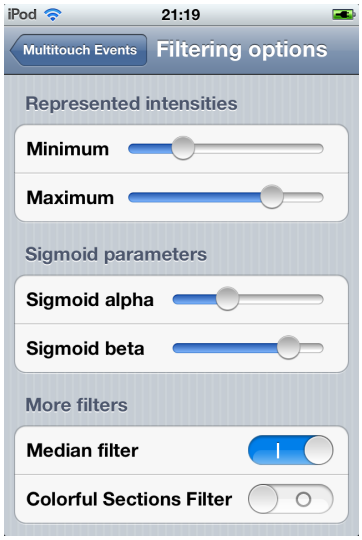


Figura 5.5.1: Vista de la pantalla d'opcions pels filtres.

Els dos primers *sliders* serien per seleccionar les intensitats que farien d'interval del *Binary Threshold Filter*, així com intensitats mínima i màxima del *Sigmoid Image Filter*. Els valors possibles estan limitats dins del rang 0-255 (ja que el màxim valor de la intensitat representable amb un *unsigned char* és 255).

Els següents dos *sliders* serien per donar valor als paràmetres alfa i beta de l'equació de la funció sigmoide. Els valors possibles són de 0 a 5, valors que se'n desconeix l'efecte, però que segons les gràfiques exemple poden ser prou bons.

Finalment, els dos botons permetrien activar o desactivar el filtre pel soroll, el *Median Filter* i el filtre personalitzat, al que hem anomenat *Colorful Sections Filter*.

6. Planificació real i cost econòmic

6.1 Planificació temporal

En aquest apartat es mostra com s'ha desenvolupat realment el projecte al llarg del temps.

Primer de tot, s'adjunta la llista de taques ordenades en el temps:

	Nombre	Inicio	Fin
1	Definició de l'àmbit del projecte	31/01/2012	12/02/2012
2	Tutorials d'Xcode i vídeos d'iTunes U d'Stanford	13/02/2012	26/02/2012
3	Primera implementació de MainViewController	27/02/2012	04/03/2012
4	**Inclusió de la llibreria ITK al projecte**	**05/03/2012**	**18/03/2012**
5	Inclusió de la versió 3.2	05/03/2012	11/03/2012
6	Intent d'incloure la versió 4.0 sense èxit	12/03/2012	18/03/2012
7	**Primers intents de representar en 3D la imatge**	**19/03/2012**	**11/04/2012**
8	Lectura exitosa d'imatges PNG i JPEG	19/03/2012	01/04/2012
9	Lectura exitosa de la imatge mèdica	02/04/2012	08/04/2012
10	Representació en 2D utilitzant la segona guia d'ITK en iOS	09/04/2012	11/04/2012
11	**Recerca de mètodes alternatius a glBegin i glEnd**	**09/04/2012**	**03/06/2012**
12	Pintant la geometria utilitzant VAO + VBO	09/04/2012	22/04/2012
13	Filtre ITK	25/04/2012	29/04/2012
14	ITK Mesh	07/05/2012	13/05/2012
15	Més alternatives	14/05/2012	03/06/2012
16	**Tècniques de visualització**	**18/04/2012**	**26/04/2012**
17	Gestos tàctils	18/04/2012	21/04/2012
18	Moviments dispositiu	22/04/2012	26/04/2012
19	Investigació sobre shaders	04/06/2012	08/06/2012
20	Començament de l'ús de la màquina virtual amb Linux	09/06/2012	09/06/2012
21	**Investigació de textures**	**10/06/2012**	**27/06/2012**
22	Varis tipus de textures descartats	10/06/2012	17/06/2012
23	**Textura 2D**	**17/06/2012**	**27/06/2012**
24	Representació textura 2D amb el cub de colors	17/06/2012	22/06/2012
25	Investigació problemes de precisió	23/06/2012	27/06/2012
26	Mostra d'un tall de la imatge canviable amb botons	17/06/2012	22/06/2012
27	**Tècniques de visualització**	**23/06/2012**	**26/04/2013**
28	Múltiples passades amb un sol pla + blending + rotacions amb ITK	23/06/2012	29/06/2012
29	Volume rendering	30/06/2012	26/04/2013
30	Començament internship	25/09/2012	25/09/2012
31	Incorporació al món laboral	28/01/2013	28/01/2013
32	Expiració llicència iOS	06/02/2013	06/02/2013
33	Polint la imatge amb filtres Threshold i Binary Threshold	26/04/2013	10/05/2013
34	Redacció de la memòria	11/05/2013	17/06/2013
35	Setmana de vacances	18/05/2013	26/05/2013
36	Més dies de festa	17/06/2013	18/06/2013
37	Implementació del controlador i la vista d'opcions pels filtres	10/06/2013	14/06/2013

Figura 6.1.1: Llista de tasques

Com es pot observar, es va tardar una mica bastant a decidir l'àmbit del projecte. Això és així perquè inicialment s'havia plantejat un projecte totalment diferent però el tutor del present va fer una altra proposta que va esdevenir l'actual projecte.

També es pot observar que les diferents tècniques de visualització, tot i que s'han anat modificant al llarg del temps, es van implementar principalment molt abans de fins i tot començar a intentar representar la imatge en 3D.

La tasca 26 no s'ha comentat al llarg del projecte, però va consistir en provar que la funció *getSlice* sí comentada funcionés. Per a fer-ho, es mostrava un tall de la imatge, i a partir d'un parell de botons afegits a la *navigation toolbar*, es sumava o restava 1 al talla mostrar, de manera que es mostrés el següent. Això, com que funcionava bé, va descol·locar encara més quan tot el muntatge amb el *volume rendering* no funcionava.

Hi ha uns punts importants a tenir en compte, ja que van ser els imprevistos amb les conseqüències més perjudicials al projecte:

1. Pels voltants del 9 de juny del 2012, es va veure la necessitat d'utilitzar la màquina virtual amb Linux per poder seguir endavant amb el desenvolupament del projecte. Tot i que no està directament representat en aquesta llista de classes, també cal comptar el temps per adaptar la pràctica de l'assignatura de gràfics per poder-la utilitzar per fer les proves necessàries. Això va suposar un temps considerable no previst.
2. El 25 de setembre del 2012, es va començar un període de pràctiques, o *internship*, a l'empresa Softonic. Aquestes pràctiques, en principi, havien de suposar tenir els matins ocupats, però les tardes lliures i els caps de setmana també. Realment no va ser així, la quantitat de feina i exigència en la qualitat era prou gran per tenir la majoria de les tardes plenament ocupades i part del cap de setmana. Tot i així, es dedicava tot el temps que pogués restar per intentar dur a terme el projecte.
3. Un cop acabat l'*internship*, i amb un curt període de descans, el 28 de gener del 2013, es va entrar a treballar a jornada completa a Softonic. Igual que abans, això suposava molt menys temps disponible per a la dedicació del projecte, però, irònicament, al no tenir feina per casa suposava un cert augment en aquest temps.
4. El 6 de febrer del 2013, la llicència que havia concedit la universitat per desenvolupar per iOS va expirar. Tot i que en aquell període s'havia utilitzat la màquina virtual de Linux pràcticament abandonant la plataforma original, això va suposar un problema que no va ser immediat de solucionar.

El problema més greu del projecte és que els problemes de gràfics importants van sorgir un cop s'havia acabat el quadrimestre, a l'estiu, i que a sobre va arribar l'*internship* com a imprevist. Un cop començat aquest *internship*, es va perdre tota possibilitat de contactar amb cap professor de gràfics que pogués ajudar, ja que el problema que es presentava requeria una llarga explicació i això es feia impossible de comentar per email: requeria trobades presencials que no es podien fer perquè no es disposava del temps per a realitzar-les.

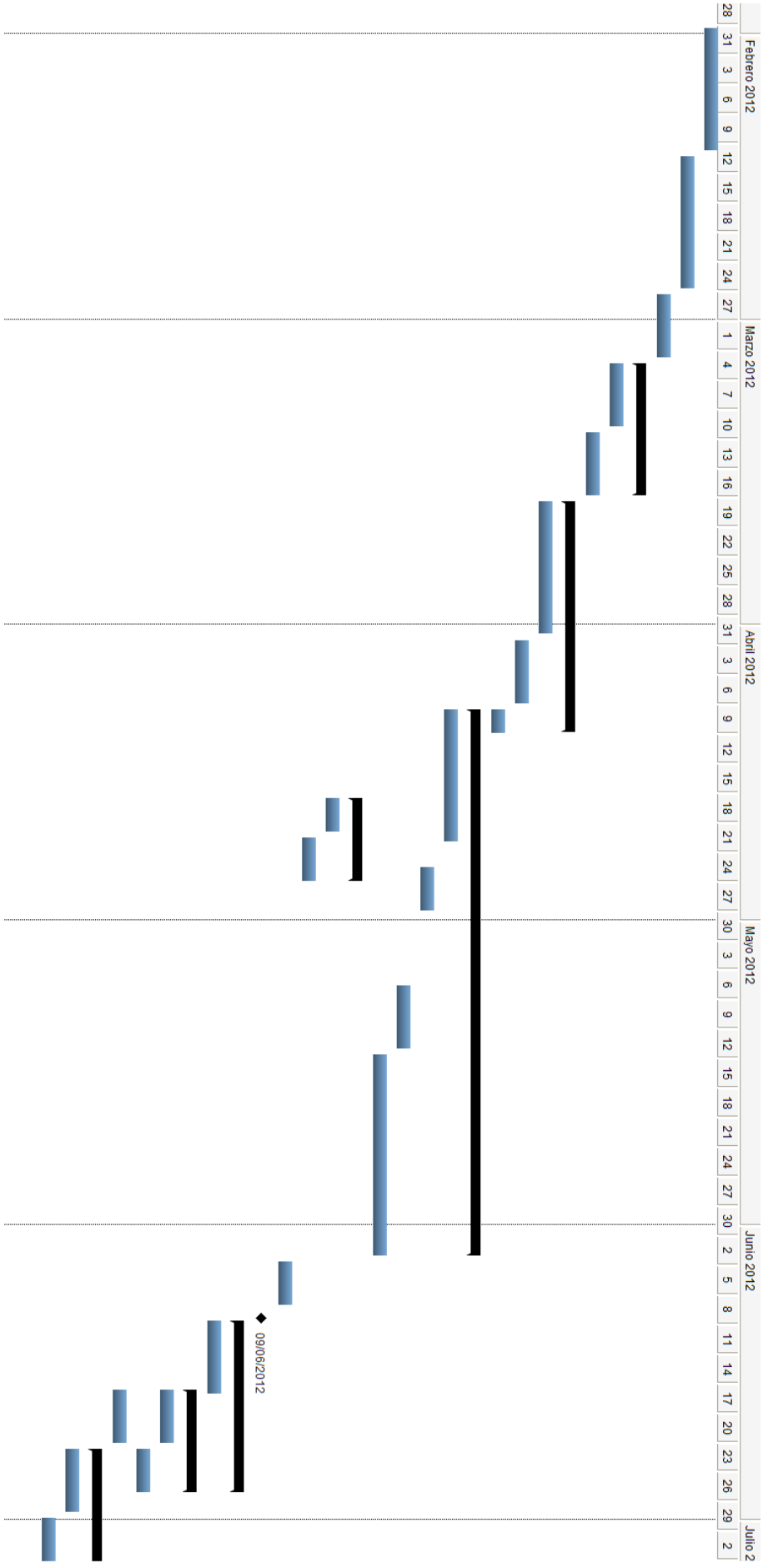
Un altre problema derivat de l'*internship* va ser que no es va poder acabar el projecte a temps durant la primera matrícula, amb el que es va rematricular i aquest cop s'havia d'acabar sí o sí.

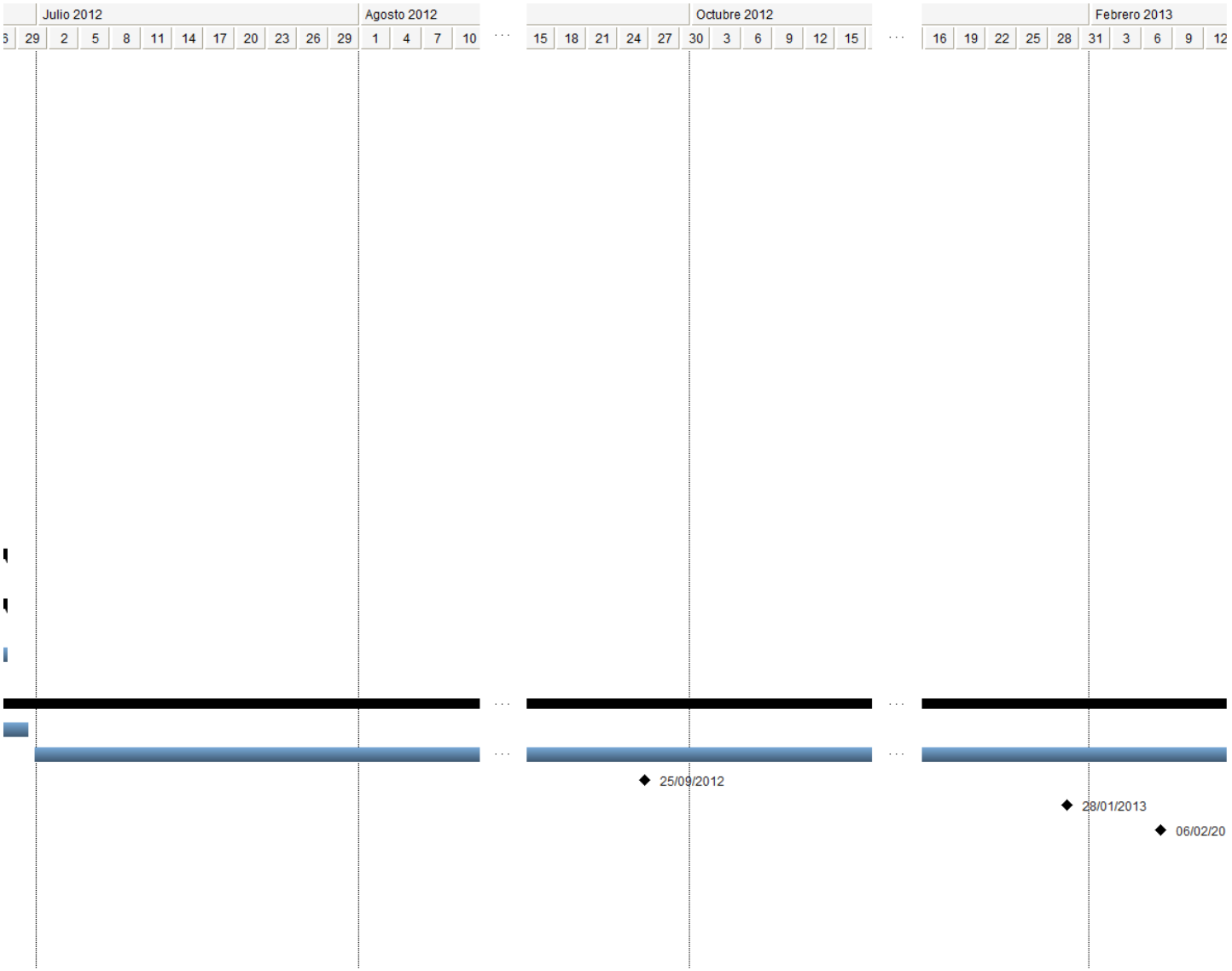
Un altre gran problema va ser la pèrdua del contacte amb el director del projecte, el qual era una figura clau en el desenvolupament total del projecte. Originalment, aquest projecte va ser proposat per ell, i jo, entusiasmat, el vaig acceptar sense pràcticament analitzar-ne els riscos i el que comportava. Tampoc tenia les bases necessàries i ho hagués hagut de preveure.
D'haver mantingut la figura de guia durant tot el desenvolupament, sense interrupcions, potser el resultat final hagués estat molt millor. Quan per fi es va recuperar el contacte, el director va fer tot el que estava a les seves mans per ajudar en la finalització del projecte (entre altres coses renovar la llicència per programar per iOS), però amb el temps que restava, es va fer el que es va poder.

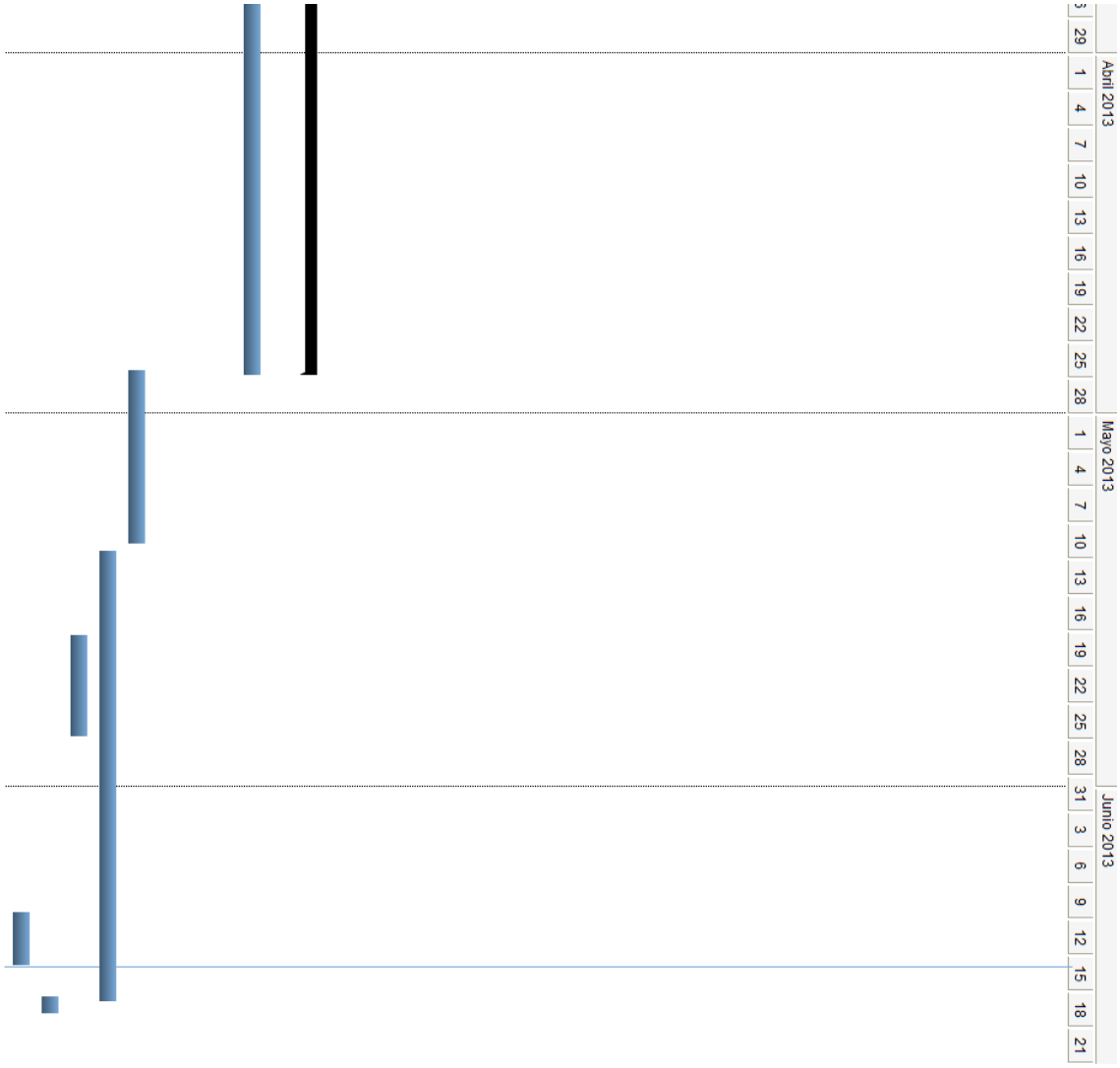
Finalment, per la redacció d'aquest document, com que es veia necessari fer-ho el millor possible, ja que en quant a resultats no es podia oferir gaire, es va haver de demanar una setmana i dos dies de vacances a Softonic, ja que no es veia manera de poder redactar un document d'aquesta magnitud si s'havia de fer amb pressa. Per això, malgrat originalment s'havien estimat dues setmanes per a redactar-lo, en realitat s'ha tardat més d'un mes.

Amb tot, al final s'ha dedicat aproximadament **1 any i 4 mesos**, dels quals es calcula una dedicació d'aproximadament **1.200 hores**, per intentar desenvolupar el projecte. Per això s'ha decidit que era hora de presentar-ne els resultats, encara que fossin negatius. No es podia allargar més la vida d'aquest projecte, resultaria absurd i probablement inútil, ja que com ja s'ha explicat anteriorment, el problema no ha estat només el temps, sinó els recursos.

La representació en format diagrama de Gantt de les tasques comentades es pot veure a continuació:


Febrero 2012
Marzo 2012
Abril 2012
Mayo 2012
Junio 2012
Julio 2
09/06/2012


Julio 2012
Agosto 2012
Octubre 2012
Febrero 2013
29 2 5 8 11 14 17 20 23 26 29 1 4 7 10
15 18 21 24 27 30 3 6 9 12 15
16 19 22 25 28 31 3 6 9 12
25/09/2012
28/01/2013
06/02/20


29
1
4
7
10
13
16
19
22
25
28
Abril 2013
1
4
7
10
13
16
19
22
25
28
31
Mayo 2013
3
6
9
12
15
18
21
Junio 2013

6.2 Eines imprevistes

A part de les eines descrites a la secció 1.4.2, algunes eines extres van haver de ser utilitzades per tal de desenvolupar el projecte. Concretament, com s'ha dit anteriorment, va fer falta l'ús d'una màquina virtual amb Linux. Per tant, a les ja descrites cal afegir:

- VMware Fusion versió 4.10.
- Distribució Ubuntu 11.10.

En aquesta distribució de Linux, fent ús de trossos de codi de pràctiques realitzades en alguna assignatura de gràfics de la universitat, és on es van desenvolupar els *shaders* i gran part del codi OpenGL que després va haver de ser traduït a OpenGL ES. Recordem que això es va haver de fer així perquè els *shaders* a executar excedien les capacitats del dispositiu físic, l'iPod Touch 4G.

Una altra eina que es podria considerar com a imprevista és l'iTunes U, l'aplicació que permet distribuir coneixement, a través de cursos o vídeos, de forma gratuïta. Concretament es va utilitzar per veure els vídeos introductoris a l'Objective-C.

6.3 Anàlisi econòmica global i comparada amb alternatives

El cost d'aquest projecte es limita a dues parts:

- El cost de la llicència de programador per dispositius iOS.
- El cost del desenvolupador del projecte amb un sol programador: la meva persona.

El cost del primer és el d'una llicència "iOS Developer Program", que equival a un preu de 99 dòlars a l'any. Tenint en compte que s'ha hagut de renovar la llicència caldria comptar 2 anys.

Cal dir que, en teoria, per desenvolupar aquest projecte s'ha utilitzat un tipus de llicència "iOS University Program", la qual és gratuïta, amb la condició de que no hi ha d'haver ànim de lucre i que tot l'esforç es dediqui a l'intent d'ensenyar les eines d'Apple. Tot i així, com que la intenció és tractar el projecte com si d'un real es tractés, s'inclou el cost de la llicència.

Per calcular el cost del desenvolupament per part del programador es calcula primer el nombre d'hores dedicades. El temps dedicat total és d'1 any i 4 mesos, però:

- D'aquest temps, els primers aproximadament 8 mesos es van poder dedicar plenament. Es calcula una mitjana de 30h setmanals dedicades abans de l'arribada de l'estiu (primers mesos). Durant aquest, al perdre el contacte amb el director es va reduir molt el treballs fins a 14h setmanals, fins finals de setembre, quan es va començar l'*internship*.
- Els següents 8 mesos es va poder dedicar molt menys al projecte, degut a l'*intership* i a la inserció en el món laboral. Es calcula una mitjana de 10h setmanals durant l'*internship* i 20h setmanals durant l'entrada a la feina.

Si comptem com a sou de programador junior, incloent els dos àmbits tractats (programador mòbil i de gràfics), com a 10€/h, el resultat final és:

$$4{,}5(mesos\ plenament\ dedicats) * 4(setmanes) * 30\left(\frac{hores}{setmana}\right) * \frac{10€}{hora} = 5.400€$$

$$3{,}5(estiu\ fins\ setembre) * 4(setmanes) * 14\left(\frac{hores}{setmana}\right) * \frac{10€}{hora} = 1.960€$$

$$4(mesos\ internship) * 4(setmanes) * 10\left(\frac{hores}{setmana}\right) * \frac{10€}{hora} = 1.600€$$

$$4(mesos\ treballant) * 4(setmanes) * 20\left(\frac{hores}{setmana}\right) * \frac{10€}{hora} = 3.200€$$

$$74€\ (\sim 99\ dòlars\ de\ llicència) * 2 = 148€$$

$$5.400€ + 1.960 + 1.600€ + 3.200€ + 148€ = \mathbf{12.308}€$$

Ara analitzem breument una possible alternativa:

Al llarg del projecte, s'ha vist que totes aquelles aplicacions i estudis realitzats sobre aquest àmbit estaven formats per un equip de programadors especialistes en gràfics conjuntament amb un o un equip de radiòlegs que els assessorava amb el tema de com estaven formades les imatges mèdiques resultants dels escàners.

Per tant, l'alternativa que es proposa per el desenvolupament del projecte és formar un equip amb:

- Un programador especialista en gràfics (programar la GPU), que cobraria al voltant de 20€/hora. Seria l'encarregat de desenvolupar els *shaders* i tota l'estructura més gràfica de l'aplicació. És important que tingués els coneixements necessaris per fer un *shader* bo i eficient, per això un junior no serviria.
- Un programador junior per iOS, que cobraria uns 10€/hora. Seria l'encarregat de muntar l'estructura de l'aplicació i desenvolupar tot el sistema de notificacions entre controladors i vistes. En principi, per la dificultat en la part d'Objective-C, hauria d'haver suficient amb un junior. Seria supervisat per un programador sènior però no es contempla el seu sou ja que es considera que la seva participació seria mínima.
- Un radiòleg que assessorés i a l'equip en relació com estan formades les imatges i com tractar-les a l'hora d'accedir a elles, aplicar-los filtres, etc. Els radiòlegs cobren al voltant de 25€/hora.

Ara bé, amb aquest nou equip, no caldria ni molt menys tant temps per desenvolupar el projecte.

Amb aquest nou equip, les 15 setmanes previstes inicialment hauria de ser de sobres per poder desenvolupar aquest aplicació. Però per donar marge a imprevistos, no es redueix aquest temps.

El programador per iOS treballaria durant les 15 setmanes, mentre que el programador gràfic i el radiòleg només en treballarien 8 d'aquestes, aproximadament. Per tant els costos serien:

$$15(setmanes) * 40\left(\frac{hores}{setmana}\right) * \frac{10€}{hora} = 6.000€$$
$$8(setmanes) * 40\left(\frac{hores}{setmana}\right) * \frac{20€}{hora} = 6.400€$$
$$8(setmanes) * 40\left(\frac{hores}{setmana}\right) * \frac{25€}{hora} = 8.000€$$

A més, cal contemplar la llicència d'iOS, però aquest cop només una: 74€.

En total, doncs:

$$6.000€ + 6.400€ + 8.000€ + 74€ = \mathbf{20.474}€$$

Noti's que el preu és considerablement més alt que el que el cost de desenvolupar-lo jo mateix. Tot i així, amb un equip com el proposat, el projecte molt probablement sí tingués èxit, no fracassaria. I no només això, sinó que tardaria moltíssim menys a 1 any i 4 mesos a desenvolupar-se.

7. Concordança de resultats i objectius

7.1 Resultats obtinguts

Una manera de mesurar l'èxit d'un projecte és utilitzar 3 paràmetres anomenats "les 3 Us". Aquestes defineixen el següent que un projecte ha estat exitós quan:

- **Útil:** El projecte aporta una certa utilitat, normalment degut a un bon anàlisi previ, un bon enfoc i un bon disseny funcional.
- **Utilitzable:** La implementació del projecte és sòlida. La infraestructura utilitzada és l'adequada, les dades estan ben tractades i emmagatzemades i el disseny arquitectònic és l'adequat: el projecte té qualitat.
- **Utilitzat:** Que el projecte estigui ben dissenyat i implementat pot semblar que implica directament l'èxit del projecte, però no és així. Es pot considerar que un projecte ha estat exitós en el moment que se n'utilitza el resultat. En el moment que hi ha usuaris interessats en ell, hi ha un sistema de suport per ells, fins i tot suggeriments de millora, en aquest punt es pot dir que el projecte ha estat un èxit rotund.

Malauradament, en el cas d'aquest projecte, com hem anat veient, no s'ha assolit ni tant sols la segona fase.

Així doncs, es pot considerar que el projecte ha fracassat? Doncs malauradament sí.

Per veure què pot haver-lo fet fracassar, fem una reflexió de perquè solen fracassar els projectes:

Materials:

- El hardware es troba sempre en constant evolució. Es podria dir que en 4 anys pràcticament queda obsolet.
- El software utilitzat de base està en constant evolució i adaptació.

Recursos:

- Les eines estan sempre també en constant adaptació tècnica i funcional.
- Les persones som poc estables, estem en constant evolució.

Objectius:

- Les funcionalitats d'un projecte solen variar durant la vida d'aquest. Si no per part dels encarregats a dur-lo a terme, que detecten possibles millores i pensen que les poden afegir al producte final sense tenir en compte el cost que pot suposar, els clients també poden voler aprofundir en una funcionalitat ja especificada, donant-li més opcions o profunditat del que inicialment semblava.
- A mesura que el temps va passant, les prestacions "exigides" pels clients augmenten. Ells no són conscients del cost que pot implicar el que per ells és un canvi nimi.

En general però, l'enemic de tot projecte, i més en el món de la informàtica, és el temps. Molts projectes fracassen per superar el temps i moments planejats originalment.
Així doncs, per què ha fracassat aquest projecte?

Segons els criteris exposats anteriorment:

El problema principal no ha sigut de hardware, tot i que l'iPod Touch no permetia córrer molts dels *shaders* i s'hagués de recórrer a la màquina virtual amb Linux. I de fet, l'iPod, al ser d'ús personal, va notar-se-li un cert deteriorament en les capacitats. Es tancaven aplicacions que abans no ho feien i es penjava més sovint. Cal dir que aquests errors també podrien produir-se per les pròpies aplicacions, sobretot si s'actualitzaven i canviaven, però això justificaria una o dues, no la majoria. Quan tantes fallen potser sí que alguna cosa hi té que veure el propi dispositiu.

Tampoc era un problema del software, ja que per molta evolució que hi hagués, es van evitar totes les actualitzacions que van sortir. Tot i així va haver un petit imprevist quan, sense voler, es va actualitzar el firmware de l'iPod a la versió 5.1.
Simplement va comportar una actualització de l'Xcode de la versió 4.2 a la versió 4.3.1, però no va tenir cap conseqüència més. Si s'hagués pogut fer servir el simulador, s'hauria pogut seguir fent servir el que simulava firmware 5.0, ja que es deixava escollir entre iPhone/iPad amb firmware 5.1 o 5.0.

El que sí va ser un problema van ser els recursos, concretament, el desenvolupador: jo. El problema va venir per una manca d'un bon anàlisi previ que pogués donar una idea del que podria suposar desenvolupar un projecte d'aquestes característiques. Es va veure una bona oportunitat i es va començar massa ràpid a intentar desenvolupar, sense analitzar si seria possible.

D'haver fer un millor anàlisi, s'hagués detectat que, per començar, no es tenia ni idea de programar per mòbil. Se sabia que hi havia certes restriccions dels recursos, sobretot la memòria, però no s'havia tocat res de mòbil abans. I per si fos poc, la programació per iOS seria amb un nou llenguatge, Objective-C, del qual no se'n sabia res. Tot i que això, per sí sol, no era un problema. Al barrejar-ho amb l'OpenGL ES és quan van sorgir els problemes.

Es tenia un nivell més aviat bàsic d'OpenGL (el propi de l'assignatura bàsica de gràfics de la universitat), i s'hauria d'haver previst que el nivell que requeriria d'OpenGL no es tenia ni de bon tros. De fet, la gran majoria de documents consultats són resultats de tesis doctorals i de projectes realitzats per equips especialitzats en la programació gràfica conjuntament amb radiòlegs que els assessoraven. El no haver pogut arribar al nivell necessari ha sigut un element clau en el fracàs d'aquest projecte.

Cal pensar que el *volume rendering* s'ha de fer per cada frame de l'aplicació, no és quelcom que es pugui fer un cop i mantenir-ho en memòria cau, per això l'eficiència era extremadament necessària i, si no es tenen els coneixements suficients, no es pot assolir.

També s'hagués pogut estudiar abans les diferències entre l'OpenGL complet i el subconjunt OpenGL ES, ja que moltíssimes limitacions i problemes van sorgir a partir d'aquesta limitació en quan a instruccions d'OpenGL disponibles.

I no només la limitació d'instruccions, sinó que moltes de les instruccions d'OpenGL normal permetien abstraure el funcionament d'OpenGL, mentre que al veure's forçat a utilitzar OpenGL ES, moltes d'aquella abstracció es perd i és necessari entendre el funcionament d'OpenGL quasi a la perfecció per poder aplicar qualsevol de les seves funcionalitats, per bàsica que sembli.

I a més, en el moment que es van haver d'incloure *shaders*, tampoc es sabia res de GLSL, i, altre cop, el nivell requerit no era precisament bàsic.

I per si tot això fos poc, a sobre calia treballar amb una llibreria externa en un tercer llenguatge, C++, del qual es tenien coneixements però res sobre el tema de *templates* que utilitza la llibreria ITK.

A tot això cal sumar-li que la guia d'ITK era per la versió 2.4 (no n'hi havia de més recent) i que pràcticament tota la documentació que hi havia era per utilitzar-la juntament amb VTK, no OpenGL. Per això mateix, moltes de les respostes a preguntes que van anar sorgint es trobaven a fòrums i llocs web semblants. Hi havia poca documentació oficial o estava poc clara.

Per últim, durant el desenvolupament dels *shaders*, va caducar la llicència de programador per iOS. Això no va suposar un problema immediat, ja que aleshores ja s'estava utilitzant la màquina virtual de Linux, però sí va ocasionar certs contratemps.

Tot això sumat va suposar que cada pas del projecte és fes d'una manera molt mes feixuga del que s'hauria. Cada pas costava molt, perquè un cop fallava alguna cosa, en molts dels casos costava discernir en quin punt fallava. Potser no es carregava bé la imatge, potser el codi OpenGL estava malament. O potser eren els *shaders*. Masses punts insegurs.

I tots aquests problemes el que van fer va ser acabar consumint és el recurs més valuós de tots: el temps. Sobretot si es té en compte que en un dels punts més crítics del projecte es va començar a treballar 8h diàries. Tot i que ja s'havia consumit gran part de les hores que en teoria cal dedicar a un projecte de final de carrera, afegir el fet de tenir molt menys temps de dedicació (només als vespres i caps de setmana) als ja existents problemes no va ajudar gens. I el projecte ja estava matriculat per segon cop: s'havia de d'acabar sí o sí.

Com a punts a favor, cal dir que teníem com a objectiu el conèixer les limitacions de la programació per mòbil, i realment les hem palpat clarament. A més, també es va utilitzar exitosament els elements que podien modificar l'escena a través de gestos tàctils i de moviments del dispositiu. No tot no s'ha acabat complint.

Per tot plegat, tot i que varis cops durant el projecte s'han vist exemples d'aplicacions semblants que han pogut córrer en dispositius iOS com l'iPod Touch del que es disposa, el resultat d'aquest projecte és, malauradament, que no ha estat possible. Però no per l'objectiu, ni les eines, sinó per la persona que ha intentat desenvolupar-lo i la seva manca de bases al respecte. Per moltes hores que s'hi dediqués, mai no eren suficients.

7.2 Conclusions

La conclusió més evident, tenint en compte que el projecte ha fracassat, seria:

"Aquest projecte no ha servit per res. Ha sigut un pèrdua d'hores i esforç i tot per res."

Bé, si això fos un projecte real i hi hagués diners pel mig, estaria d'acord. Però tractant-se d'un projecte final de carrera, el valor acadèmic que ha aportat aquest projecte ha sigut prou important.

Com ja s'ha dit a la secció de la concordança amb els objectius plantejats, no s'ha aconseguit desenvolupar l'aplicació que en un principi es pretenia. Però, tot i així, tot i no haver arribat al nivell suficient, s'han après moltes coses. Per començar, s'ha començat a veure una mica l'Objective-C. No s'ha pogut tractar gaire perquè la major part del temps ha sigut dedicada exclusivament a OpenGL i a GLSL, però alguna cosa s'ha vist.

A més, també s'ha après OpenGL. Altre cop, potser no ha estat suficient, però s'han après moltíssimes coses que abans es desconeixien. Que si el *programmable pipeline*, que si els *shaders*, etc. De fet, durant el transcurs del projecte es va cursar una assignatura de gràfics avançada, a la qual es tractaven alguns dels temes utilitzats en aquest projecte. Tot i que en aquell moment el projecte no tenia encara prou complexitat gràfica (d'altra banda s'hagués pogut preguntar al professor corresponent algun dubte sobre OpenGL), en alguns punts el projecte i l'assignatura coincidien. De fet, es pot dir que gràcies al projecte, **es va acabar traient matrícula d'honor de l'assignatura**. Això demostra que el projecte no ha estat inútil completament, i també que ni tant sols els coneixements de l'assignatura de gràfics avançada va ser suficient per poder acabar el projecte.

La conclusió que en trec jo, doncs, és que tot i haver fracassat el projecte no ha estat del tot desaprofitat.

Ara bé, no hagués hagut de començar un projecte de tal envergadura sense haver mirat molt millor abans on em posava. Tampoc havia d'haver gestionat tant malament el temps; el fet de començar a treballar a mig projecte va ser nefast.

Hagués pogut detectar que el projecte acabaria fracassant i l'hagués hagut de descartar llavors, però en aquell moment estava cursant l'assignatura de gràfics i veia un gran ventall d'opcions que després de poc han servit. A més, era un projecte ambiciós i ja es va començar pensant que seria difícil, però no fins al punt de no assolir-lo.

A més, s'ha de tenir en compte una última cosa: aquest projecte no ha estat gens progressiu. **Des del primer moment, ha exigit un alt nivell**. Si hagués pogut presentar una estructura més progressiva, on primer s'assolissin objectius fàcils i després complicats, seria un altre tema. Però no, des d'un principi, el que es va plantejar ja era d'una certa dificultat elevada.

La conclusió final és que, tot i la ràbia i frustració de no haver-me'n pogut sortir, i el fet de pensar de que ara probablement no el tornaria a començar, ha estat una experiència valuosa. Dels errors s'aprèn, i aquest no serà una excepció. En futurs projectes, l'anàlisi previ s'haurà de fer molt millor.

7.3 Treballs futurs

Aquest apartat té poc sentit quan es tracta d'un projecte que ha fracassat com aquest, però hi ha algunes coses que se'n poden dir.

Per començar, com és evident, un treball futur podria ser acabar el que es va començar: acabar d'implementar el *volume rendering*, si és que no hi ha cap altre tècnica més eficient aleshores, i mirar de representar així la imatge en 3D. A partir de llavors, caldria decidir quins són els filtres, dels que diposa ITK, que es podrien aplicar.

Estaria bé, a més, un cops decidits els filtres, només incloure aquella part de la llibreria ITK que és necessària per aquests, ja que actualment s'adjunta sencera i no té perquè ser necessari.

A més, si el *ray casting* ja funcionés correctament, també es podria aplicar alguna de les optimitzacions proposades.

Sobre les tècniques de visualització afegides, zoom, pan i rotació, caldria potser buscar la manera implementar el pan, si realment es notés necessari un cop representades les imatges en 3D. A més, seria una bona idea fer que el giroscopi es guardés l'estat en que es troba el dispositiu just en el moment de clicar l'opció d'utilitzar-lo i basés les rotacions dels angles des d'aquest estat inicial, en comptes de forçar que l'estat inicial sigui el del dispositiu pla, recolzat sobre una superfície.

Finalment, caldria permetre la representació de diferents imatges, ja que actualment es fa servir sempre la mateixa. Caldria d'alguna manera proporcionar la possibilitat d'obtenir imatges, ja que directament no es poden incloure en el dispositiu com si d'un disc dur es tractés. Potser caldria allotjar-les en un servidor remot i llavors descarregar cada vegada la imatge que es vulgui representar. Compilar una aplicació personalitzada amb les imatges resulta òbviament impensable, així que algun mètode per obtenir-les caldria buscar. I si es baixessin d'un servidor remot, a més implicaria temes de seguretat, ja que la confidencialitat de la informació d'un pacient és una prioritat.

Això últim implicaria canvis importants en el codi actual, ja que moltes coses estan "*hardcodejades*", en el sentit que són totalment dependents de la imatge actual.

Caldria adaptar al codi a una versió més genèrica que pogués acceptar qualsevol tipus d'imatge de qualsevol mida.

Bibliografia

Especificacions dels dispositius

- http://www.iphoneheat.com/2010/09/ipod-touch-4g-specs/
- http://en.wikipedia.org/wiki/Timeline_of_Apple_Inc._products
- http://www.gsmarena.com/apple_iphone_4-3275.php
- http://www.gsmarena.com/apple_iphone_4s-4212.php
- http://www.gsmarena.com/apple_ipad_2_wi_fi_+_3g-3848.php

Objective-C

- http://www.techotopia.com/index.php/The_Basics_of_Object_Oriented_Programming_in_Objective-C
- http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/AppDesignBasics/AppDesignBasics.html
- http://developer.apple.com/library/ios/#documentation/uikit/reference/UIApplicationDelegate_Protocol/Reference/Reference.html
- http://developer.apple.com/library/ios/#documentation/general/conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html
- http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/AppDesignBasics/AppDesignBasics.html
- https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html
- https://developer.apple.com/technologies/mac/cocoa.html

ITK

- *ITK on the iOS* (Boris Shabash, Ghassan Hamarneh, Zhi Feng Huang, Luís Ibáñez)
- *ITK Image IO interface with Apple iOS* (Boris Shabash, Ghassan Hamarneh, Zhi Feng Huang, Luis Ibañez)
- *ITK Software Guide - Second Edition* (Luís Ibáñez, Will Schroeder, Lydia Ng, Josh Cates).
- *Getting Started with ITK* (Luís Ibáñez, William Schroeder, Insight Software Consortium)
- *ITK Registration Methods* (Kitware Inc.)
- *ITK Lecture 7 – Writing Filters, Part 1* (Damion Shelton)
- *ITK Lecture 16 – ITK Pipeline* (Dr. John Galeotti)
- *ITK Workshop – Writing a new ITK Filter*
- *ITK Basic Filters* (Kitware Inc.)
- http://www.itk.org/Wiki/ITK/Configuring_and_Building
- http://www.itk.org/Wiki/MetaIO
- http://www.kitware.com/source/home/post/13
- http://www.kitware.com/media/html/AlternativeMemoryModelsForITK.html
- http://www.itk.org/Wiki/ITK/File_Formats#File_Formats_and_Pixel_Types
- http://www.itk.org/pipermail/insight-users/2011-April/040732.html
- http://www.itk.org/pipermail/insight-users/2011-April/040789.html
- http://osirixpluginwithitk320.wikispaces.com/CompileITK4

- http://www.itk.org/pipermail/insight-users/2010-June/037400.html
- http://itk-users.7.n7.nabble.com/Help-on-image-size-td4599.html

VTK

- http://www.vtk.org/
- http://vtk.1045678.n5.nabble.com/VTK-on-iOS-td3406784.html
- http://www.vtk.org/pipermail/vtkusers/2012-January/120948.html
- http://www.cmake.org/pipermail/cmake/2011-May/044202.html
- http://comments.gmane.org/gmane.comp.lib.vtk.user/66254

Shaders – GLSL

- Apunts VA: 3.1. GLSL- OpenGL Shading Language
- OpenGL 2.0 - GLSL Shaders How To
- http://en.wikipedia.org/wiki/Shader
- http://www.opengl.org/wiki/Shader
- http://www.opengl.org/wiki/OpenGL_Shading_Language
- http://en.wikipedia.org/wiki/GLSL
- http://nehe.gamedev.net/article/glsl_an_introduction/25007/
- http://en.wikipedia.org/wiki/OpenGL_ARB
- http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_1.pdf
- http://antongerdelan.net/opengl/debugshaders.html
- http://antongerdelan.net/opengl/geometry.html
- http://www.opengl.org/wiki/Sampler_(GLSL)
- http://www.khronos.org/opengles/2_X/

VBO

- Apunts VA: 2. Pintat eficient
- https://developer.apple.com/library/ios/#DOCUMENTATION/GLkit/Reference/GLKView_ClassReference/Reference/Reference.html
- http://www.opengl.org/wiki/OpenGL_Context
- http://www.opengl.org/wiki/Buffer_Object
- http://www.opengl.org/wiki/Vertex_Buffer_Object
- http://www.arcsynthesis.org/gltut/Positioning/Tutorial%2005.html
- https://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html

Textures 3D

- https://developer.apple.com/library/ios/#documentation/GLkit/Reference/GLKTextureLoader_ClassRef/Reference/Reference.html
- http://www.opengl.org/discussion_boards/showthread.php/168662-Texture-3D
- http://content.gpwiki.org/index.php/OpenGL:Tutorials:3D_Textures
- http://www.opengl.org/sdk/docs/man/xhtml/glTexImage3D.xml

Array Textures

- http://www.opengl.org/wiki/Array_Texture

Buffer Textures

- http://www.opengl.org/wiki/Buffer_Texture
- http://www.opengl.org/sdk/docs/man3/xhtml/glTexBuffer.xml

OpenGL ES 2.0

- *OpenGL ES Programming Guide for iOS*
- http://www.songho.ca/opengl/gl_transform.html
- http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/APITables.xml
- http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf
- http://games.ianterrell.com/how-to-create-and-render-composite-objects-with-glkit/

Textures 2D

- http://www.khronos.org/opengles/sdk/1.1/docs/man/glTexImage2D.xml
- http://iphonedevelopment.blogspot.com.es/2009/05/opengl-es-from-ground-up-part-6_25.html
- http://games.ianterrell.com/how-to-texturize-objects-with-glkit/
- http://stackoverflow.com/questions/3569261/multiple-textures-in-glsl-only-one-works

Accés a textura 2D

- http://stackoverflow.com/questions/9432223/how-to-render-3d-texture-data-with-2d-textures-in-opengl-es-2-0
- http://stackoverflow.com/questions/9241583/how-can-i-use-a-3-d-texture-in-ios
- http://stackoverflow.com/questions/15769056/gl-luminance-with-byte-array-on-opengl-es-2-0
- http://www.glprogramming.com/red/chapter09.html
- http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesForWorkingWithTextureData/TechniquesForWorkingWithTextureData.html
- http://stackoverflow.com/questions/14150941/webgl-glsl-emulate-texture3d
- http://www.youtube.com/watch?v=rfQ8rKGTVlg#t=26m00s
- http://www.khronos.org/message_boards/showthread.php/7225-3D-textures-in-WebGL
- http://stackoverflow.com/questions/12772545/glsl-simulating-3d-texture-with-2d-texture-probelm
- http://stackoverflow.com/questions/15588860/what-exactly-are-eye-space-coordinates
- http://stackoverflow.com/questions/5879403/opengl-texture-coordinates-in-pixel-space
- http://hacksoflife.blogspot.com.es/2009/12/texture-coordinate-system-for-opengl.html

Blending

- http://www.opengl.org/wiki/Blending
- http://antongerdelan.net/opengl/blending.html

Alternatives a volume

- *X-Ray Casting: Fast Volume Visualization Using 2D Texture Mapping Techinques* (Youngser Park, Robert W. Lindeman, James K. Hahn).
- *Volume Rendering Strategies On Mobile Devices* (José M. Noguera, Juan-Roberto Jiménez, Carlos J. Ogáyar, Rafael J. Segura).
- https://en.wikipedia.org/wiki/Computed_tomography
- http://en.wikibooks.org/wiki/Basic_Physics_of_Nuclear_Medicine/Three-Dimensional_Visualization_Techniques

Volume rendering / ray casting

- *Real-Time Volume Graphics: GPU-Based Volume Rendering* (Christol Rezk Salama)
- *Real-Time Volume Graphics: GPU-Based Ray-Casting* (Markus Hadwiger)
- *Volumetric Methods in Visual Effects* (Magnus Wrenninge, Nafees Bin Zafar): pàgines 51-62
- *GPU Ray Casting* (Ricardo Marqués, Luís Pablo Santos, Peter Leskovsky, Céline Paloc).
- *Volume Ilustration: Non-Photorealistic Rendering of Volume Models* (David Ebert, Penny Rheingans)
- *A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting* (Simon Stegmaier, Magnus Strengert, Thomas Klein, Thomas Ertl)
- *Acceleration Techniques for GPU-based Volume Rendering* (J. Krüger and R. Westermann)
- http://en.wikipedia.org/wiki/Volume_ray_casting
- http://en.wikipedia.org/wiki/Volume_rendering
- http://stackoverflow.com/questions/9482572/volume-rendering-using-glsl-with-ray-casting-algorithm
- https://github.com/toolchainX/Volume_Rendering_Using_GLSL
- http://http.developer.nvidia.com/GPUGems3/gpugems3_ch29.html
- http://www.real-time-volume-graphics.org/
- http://www.cabiatl.com/mricro/raycast/
- http://www.daimi.au.dk/~trier/?page_id=98
- http://prideout.net/blog/?p=64
- https://www.marcusbannerman.co.uk/index.php/component/content/article/42-articles/97-vol-render-optimizations.html
- http://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm
- http://pages.cs.wisc.edu/~nowak/779/779ClassProject.html
- http://graphicsrunner.blogspot.com.es/2009/01/volume-rendering-101.html
- http://www.cs.csustan.edu/~rsc/CS3600F02/PerPixel.pdf
- http://www.visualizationlibrary.org/documentation/pag_guide_raycast_volume.html
- http://www.voreen.org/129-Ray-Casting.html
- http://www.cs.csustan.edu/~rsc/CS3600F02/PerPixel.pdf
- http://cumbia.informatik.uni-stuttgart.de/ger/research/fields/current/spvolren/

Renderitzar a una textura

- http://ogltotd.blogspot.com.es/2006/12/render-to-texture.html
- http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/WorkingwithEAGLContexts/WorkingwithEAGLContexts.html
- http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/
- http://www.opengl.org/wiki/Framebuffer_Object_Examples#Quick_example.2C_render_to_texture_.282D.29
- http://www.songho.ca/opengl/gl_fbo.html
- http://stackoverflow.com/questions/11617013/why-would-glbindframebuffergl-framebuffer-0-result-in-blank-screen-in-cocos2d
- http://stackoverflow.com/questions/2213030/whats-the-concept-of-and-differences-between-framebuffer-and-renderbuffer-in
- http://kineplay.com/ben/?p=810
- http://www.idevgames.com/forums/thread-8054.html
- http://stackoverflow.com/questions/4900799/how-to-switch-between-rendering-and-presenting-framebuffers-in-iphone-opengl-es
- http://stackoverflow.com/questions/12600023/how-to-read-pixels-from-a-rendered-texture-in-opengl-es
- http://www.idevgames.com/forums/thread-1785.html
- http://iphonedevsdk.com/forum/iphone-sdk-game-development/32350-rendering-framebuffer-texture-not-working.html
- http://stackoverflow.com/questions/10455329/opengl-es-2d-rendering-into-image
- http://www.idevgames.com/forums/thread-1785.html
- http://stackoverflow.com/questions/4041682/android-opengl-es-framebuffer-objects-rendering-to-texture
- http://stackoverflow.com/questions/9528923/opengl-es-2-x-how-to-read-pixels-after-gldiscardframebufferext
- http://mickyd.wordpress.com/2012/05/20/creating-render-to-texture-secondary-framebuffer-objects-on-ios-using-opengl-es-2/
- http://lists.apple.com/archives/mac-opengl/2007/Nov/msg00009.html
- http://stackoverflow.com/questions/6650148/unable-to-load-multiple-textures-in-opengl-es-2-0
- http://www.gamedev.net/topic/287220-glsl---multiple-texture-passing--updated/
- http://www.opengl.org/wiki/Vertex_Transformation
- http://stackoverflow.com/questions/5879403/opengl-texture-coordinates-in-pixel-space

BSP i Octree

- http://en.wikipedia.org/wiki/Binary_space_partitioning
- http://en.wikipedia.org/wiki/Octree

Gestos tàctils

- *Touch Gesture Guide* (Luke Wroblewski)
- http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizer_basics/GestureRecognizer_basics.html

- http://developer.apple.com/library/ios/#documentation/uikit/reference/UIKit_Framework/_index.html
- http://www.raywenderlich.com/6567/uigesturerecognizer-tutorial-in-ios-5-pinches-pans-and-more

Moviments del dispositiu

- http://es.wikipedia.org/wiki/Aceler%C3%B3metrohttp://www.diferenciaentre.net/la-diferencia-entre-giroscopo-y-acelerometro/
- http://es.wikipedia.org/wiki/Gir%C3%B3scopo
- http://developer.apple.com/library/ios/#documentation/GLkit/Reference/GLKMatrix4/Reference/reference.html
- http://developer.apple.com/library/ios/#documentation/CoreMotion/Reference/CMMotionManager_Class/Reference/Reference.html
- http://developer.apple.com/library/ios/#documentation/CoreMotion/Reference/CMAttitude_Class/Reference/Reference.html#//apple_ref/occ/cl/CMAttitude
- http://developer.apple.com/library/ios/#documentation/CoreMotion/Reference/CMDeviceMotion_Class/Reference/Reference.html#//apple_ref/occ/cl/CMDeviceMotion
- http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Introduction/Introduction.html
- http://stackoverflow.com/questions/3245733/apple-gyroscope-sample-code
- http://stackoverflow.com/questions/11877025/arc-forbids-explicit-message-send-of-retain-issue
- http://www.opengl.org/discussion_boards/showthread.php/165806-Determining-angles-from-Modelview-Matrix

Filtres

- http://en.wikipedia.org/wiki/Sigmoid_function
- http://stackoverflow.com/questions/6800355/uitableview-grouped-iphone-development
- http://adeem.me/blog/2009/05/20/iphone-sdk-tutorial-part-3-grouped-uitableview/
- http://www.iphonesdkarticles.com/2009/01/uitableview-sectioned-table-view.html
- http://stackoverflow.com/questions/5752930/how-to-set-group-tables-section-header-text
- http://stackoverflow.com/questions/10505708/how-to-set-the-uitableview-section-title-programmatically-iphone-ipad
- http://stackoverflow.com/questions/11196734/creating-grouped-uitableview-with-different-cell-types
- http://stackoverflow.com/questions/1802707/detecting-which-uibutton-was-pressed-in-a-uitableview
- http://stackoverflow.com/questions/9429618/adding-cells-programmatically-to-uitableview
- http://www.iosdevnotes.com/2011/10/uitableview-tutorial/
- http://iphonedevsdk.com/forum/iphone-sdk-development/97423-programmatically-build-uitableview-sections-populate-inner-cells.html

- http://stackoverflow.com/questions/8836563/add-button-to-uitableviewcell
- http://stackoverflow.com/questions/812426/uitableview-setting-some-cells-as-unselectable
- http://stackoverflow.com/questions/7799222/how-to-make-a-uitableviewcell-unselectable
- http://stackoverflow.com/questions/16174018/slide-uislider-programmatically
- http://stackoverflow.com/questions/11464811/assign-method-to-uislider-programmatically-using-objective-c
- http://www.mysamplecode.com/2013/01/ios-programmatically-create-uislider.html
- http://stackoverflow.com/questions/7302169/selector-function-with-int-parameter
- http://stackoverflow.com/questions/2297613/selector-with-multiple-arguments
- http://stackoverflow.com/questions/8348626/get-value-from-a-uislider-inside-a-cell
- http://developer.apple.com/library/ios/#documentation/uikit/reference/UISwitch_Class/Reference/Reference.html
- http://developer.apple.com/library/ios/#documentation/UIKit/Reference/UISlider_Class/Reference/Reference.html
- http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/tableview_iphone/tableviewstyles/tableviewcharacteristics.html

Integració d'ITK a Linux

- http://www.itk.org/pipermail/insight-users/2007-March/021537.html
- http://www.itk.org/pipermail/insight-users/2007-November/024358.html
- http://www.itk.org/pipermail/insight-users/2012-August/045503.html
- http://www.itk.org/pipermail/insight-users/2009-April/029795.html
- http://lists.trolltech.com/qt-interest/2007-01/thread00356-0.html
- http://qt-project.org/quarterly/view/using_cmake_to_build_qt_projects
- http://www.cmake.org/cmake/help/cmake_tutorial.html
- http://www.cmake.org/cmake/help/syntax.html
- http://doc.qt.digia.com/qt/qmake-manual.html
- http://doc.qt.digia.com/qt/qmake-tutorial.html
- http://doc.qt.digia.com/qt/qmake-variable-reference.html
- http://qt-project.org/forums/viewthread/7559
- http://www.itk.org/pipermail/insight-users/2012-July/045427.html
- http://www.cmake.org/pipermail/cmake/2006-April/008920.html
- http://www.itk.org/pipermail/insight-users/2007-July/022912.html -fins- http://www.itk.org/pipermail/insight-users/2012-April/044425.html

Referències extres

[1]: http://healthland.time.com/2010/10/15/is-your-touch-screen-cleaner-than-a-toilet-flusher/
[2]: http://en.wikipedia.org/wiki/List_of_open-source_healthcare_software
[3]: http://www.imaginaformacion.com/tutoriales/primera-aplicacion-iphone-hola-mundo/
[4]: http://www.youtube.com/watch?v=T6FMzueNMRs
[5]: http://www.raywenderlich.com/3664/opengl-es-2-0-for-iphone-tutorial
[6]: http://www.raywenderlich.com/5223/beginning-opengl-es-2-0-with-glkit-part-1
[7]: http://www.edumobile.org/iphone/iphone-programming-tutorials/opengl-application-in-iphone/
[8]: http://iphonedevelopment.blogspot.com.es/2009/05/opengl-es-from-ground-up-table-of.html
[9]: http://maniacdev.com/2011/05/opengl-es-tutorials-beginner-through-advanced/
[10]: http://www.opengl.org/wiki/Framebuffer
[11]: http://www.opengl.org/wiki/Buffer_Object#General_use
[12]:http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/BestPracticesforShaders/BestPracticesforShaders.html#//apple_ref/doc/uid/TP40008793-CH7-SW3
[13]: http://unixhelp.ed.ac.uk/CGI/man-cgi?od
[14]: http://es.wikipedia.org/wiki/Coordenadas_homogéneas
[15]: http://songho.ca/math/homogeneous/homogeneous.html

Annex 1

Sintaxi GLSL

La funció principal d'un *shader* és la típica *void main*. Un *main* d'un *shader* pot fer moltes coses, però el més normal és que faci, com a mínim, l'equivalent al *fixed pipeline* en aquell punt del procés.

Variables

Tipus

Les variables poden ser:

- Dels tipus escalars *float*, *int* o *bool*.
- Del tipus de vectors *vec2*, *vec3* o *vec4*, de 2, 3 o 4 components respectivament.
- Del tipus de matrius *mat2*, *mat3*, *mat4* (matrius de 2x2, 3x3 i 4x4 respectivament).
- Del tipus *sampler1D*, *sampler2D* o *sampler3D* (textures de 1D, 2D o 3D respectivament).

A més, també es poden construir estructures (*structs*) de la forma habitual:

```
struct str {
      vec3 field1;
      float field2;
};
```

Una definició d'un *struct* defineix implícitament un constructor:

```
str(f1, f2); //Crearia una estructura str assignant f1 a field1
i f2 a field2.
```

Per accedir a un vector de 4 components es pot fer servir:

- .x, .y, .z i .w, si el vector representa un punt o un vector.
- .r, .g, .b i .a, si el vector representa un color.
- .s, .t, .p, .q, si el vector representa coordenades de textura.

El *shader* no comprova que s'accedeixi amb la component corresponent al tipus que representa la variable, ja que no té manera de distingir entre un vector que representa un punt o un que representa un color. El fet de distingir entre .x i .r, per exemple, és només per facilitar la llegibilitat del codi.

La inicialització de matrius es fa per columnes:

mat2 m = mat2(1.0, 2.0, 3.0, 4.0);

Donarà de resultat:

$$m = \begin{pmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{pmatrix}$$

Qualificadors

Cada variable pot ser del tipus *attribute*, *uniform*, *varying*, *const* o de cap d'aquests tipus.

Les variables tipus ***attribute*** són passades per l'usuari des de l'aplicació al *vertex shader*.

Poden variar per vèrtex.

Es declaren en àmbit global i són de només lectura.

Només poden ser de tipus *float*, *vec2*, *vec3*, *vec4*, *mat2*, *mat3* o *mat4*.

Les variables tipus ***uniform*** són passades per l'usuari des de l'aplicació fins al *vertex* o *fragment shader*.

Poden variar per primitiva.

Es declaren en àmbit global i són de només lectura.

Poden ser de qualsevol tipus.

Les variables tipus ***varying*** són variables que passen del *vertex shader* al *fragment shader*.

Es declaren en àmbit global.

Són de sortida pel *vertex shader* i d'entrada pel *fragment shader*, això vol dir que el valor que arriba al *fragment shader* és una interpolació dels valors que pren la variable en els vèrtexs implicats en aquell fragment.

Les variables de tipus ***const*** es poden utilitzar tant al *vertex* com al *fragment shader*.

Són de només lectura: com indica el nom, són constants.

Si una variable no té cap dels anteriors tipus, és que es tracta d'una variable local o global que limita la seva existència a l'execució en curs del *shader*, i la qual es pot tant llegir com escriure.

Operadors

Operador	Descripció
[]	Índex
.	Selector de membre (propietat)
a++ o a--	Increment/decrement postfix
++b o –b	Increment/decrement prefix
!	Negació i not lògica
*, /	Multiplicació i divisió
+, -	Suma i resta
<, >, <=, >=	Més petit que, més gran que, més petit o igual que, més gran o igual que
==, !=	Igualtat, diferència
&&	And lògica
^^	Xor lògica
||	Or lògica
=, +=, -=, *=, /=	Assignació

Funcions predefinides

GLSL disposa d'algunes funcions predefinides per facilitar al programador alguns càlculs. Alguns exemples són:

- Funcions matemàtiques: sin(), cos(), tan(), radians(), degrees(), pow(), exp(), log(), sqrt(), abs(), floor(), ceil(), etc.
- Funcions geomètriques: length(), distance(), dot(), normalize(), reflect(), etc.
- Funcions d'accés a textures: texture1D(), texture2D(), texture3D(), textureCube(), etc.