9Rubik

Simulador Rubik multi-figura en OpenGL sobre plataforma iOS

Carles Noguera i Balasch
Estudiant d'Enginyeria Tècnica en Informàtica de Sistemes

Jordi Ceballos Villach
Consultor del Treball Final de Carrera
Memòria — Gener del 2013

Índex

1 El Projecte **7**

1.1 Definició 7

1.2 Objectius 8

1.3 Eines i Tecnologia 8

2 Pla de Treball **9**

2.1 Fases del Projecte 9

2.2 Viabilitat del Projecte 9

2.3 Calendari del Projecte 10

3 Model de Negoci **11**

3.1 Composició 11

3.1.1 La geometria 11

3.1.2 Els talls 12

3.1.3 Les peces 12

3.2 Funcionament 13

3.2.1 Els plans 13

3.2.2 Els talls 17

3.2.3 Els nivells 18

3.2.4 Les peces 18

3.2.5 La solució 18

3.3 Manipulació 20

3.3.1 Movent el "Cub" 20

3.3.2 Movent les peces 20

4 Tècnica i Tecnologia **21**

4.1 Objective-C . . . 21
4.1.1 La nomenclatura . . . 21
4.1.2 La codificació . . . 21
4.1.3 Gestió d'instàncies . . . 22
4.1.4 Altres característiques . . . 23
4.2 Cocoa . . . 23
4.2.1 Prefixos . . . 23
4.2.2 Model–Vista–Controlador . . . 23
4.3 XCode i Instruments . . . 24
4.4 OpenGL . . . 24
4.5 Geometria . . . 24
4.5.1 Equació General del Pla . . . 24
4.5.2 Posició respecte el Pla . . . 24
4.5.3 Interseccións . . . 25
4.5.4 Matriu de Rotació . . . 26
4.5.5 La Càmera en l'Escena . . . 27
4.5.6 Moviment de la Càmera . . . 28
4.5.7 Sentit del Moviment . . . 29
4.6 OpenGL . . . 29
4.6.1 Aplicar la Textura . . . 29
4.6.2 Textura i Material . . . 30

5 Anàlisi i Disseny 31

5.1 Usuaris . . . 31
5.1.1 Jugador . . . 31
5.2 Casos d'Ús . . . 31
5.2.1 Mòdul de Vista . . . 32
5.2.2 Mòdul de Moviments . . . 32
5.3 Classes . . . 33
5.3.1 Vista–Controlador . . . 33
5.3.2 Model . . . 34
5.4 Prototipat . . . 36
5.4.1 Pantalla Principal . . . 36

5.4.2 Plantalla de Joc 36

6 Implementació 38

6.1 L'aplicació 38

6.1.1 RBK9AppDelegate 38

6.1.2 RBK9ListController 38

6.2 El Model 40

6.2.1 RBK9Matrix 41

6.2.2 RBK9Vector 44

6.2.3 RBK9Figure 45

6.2.4 RBK9Face 46

6.2.5 RBK9Material 47

6.2.6 RBK9Texture 48

6.2.7 RBK9Piece 48

6.2.8 RBK9Plane 49

6.2.9 RBK9Cut 51

6.2.10 RBK9Camera 51

6.2.11 RBK9Rubik 52

6.3 La Vista 54

6.3.1 La definició 54

6.3.2 Activar l'OpenGL 55

6.3.3 Configurar l'OpenGL 55

6.3.4 Dibuixar el "Cub" 55

6.3.5 Seleccionar les Peces 56

6.3.6 Configurar els Gestos 57

6.3.7 Controlar els Gestos 57

6.4 El Controlador 58

6.4.1 Definició 58

6.4.2 Accions de la Interfície 59

6.4.3 Control de l'Aparença 59

6.4.4 Dibuixat i Seleccionat 60

6.4.5 Moviment de Peces 60

6.4.6 Delegació de la Vista 60

7 Conclusió **63**

7.1 Línies Obertes 63

7.1.1 Simulador 63

7.1.2 Aplicació 64

7.2 Conclusions 65

A Captures de l'aplicació **66**

A.1 Arrancada del Programa 66

A.2 Llistat de "Cubs" 66

A.3 Seleccionar un "Cub" 67

A.4 Rotació de la càmera 67

A.5 Moure la càmera 68

A.6 Movent les peces 68

B Altres Modes **70**

B.1 Usuaris 70

B.1.1 Modelador 70

B.1.2 Dissenyador 71

B.2 Casos d'Ús 72

B.2.1 Mòdul de Modelat 72

B.2.2 Mòdul de Pintat 73

B.2.3 Mòdul de Tallat 74

B.2.4 Mòdul de Peces 74

B.3 Prototipat 75

B.3.1 Pantalla de Disseny 75

C Avaluació de Prototipus **78**

C.1 Estratificació 78

C.1.1 Perfils 78

C.1.2 Preguntes 78

C.2 Tasques 79

C.2.1 Aspectes a valorar 79

C.2.2 Tasques a realitzar 79

C.3 Preguntes 80

C.3.1 Navegar 80
C.3.2 Produir 80
C.3.3 Jugar 81
C.3.4 Girar 81
C.3.5 Dissenyar 82
C.3.6 Editar 82
C.3.7 Conclusió 83

D OpenGL **84**

D.1 OpenGL sobre iOS 84
D.2 Versions d'OpenGL ES 85
D.3 Configurant l'OpenGL ES 85
D.3.1 Creant el context 85
D.3.2 Creant els buffers 85
D.4 Capacitats de l'OpenGL ES 87
D.5 Tipus de dades 87
D.6 La Vista 87
D.6.1 Vista Ortogràfica 88
D.6.2 Vista Perspectiva 88
D.7 Dibuixat 89
D.7.1 Dibuixar Arrays 89
D.7.2 Dibuixar Elements 90
D.8 La Llum 90
D.8.1 Activar la Il·luminació 91
D.8.2 Components de color 91
D.8.3 Posició 92
D.8.4 Atenuació 92
D.8.5 Llum direccional 92
D.8.6 Les Normals 92
D.9 El Material 93
D.10 Textures 93
D.10.1 Activar les textures 94
D.10.2 Creant textures 94

D.10.3 Limitacions 94
D.10.4 Coodenades 95
D.10.5 Altres paràmetres 95
D.11 Transformacions 95
D.11.1 Carregar i desar matrius 95
D.11.2 Multiplicar Matrius 95
D.11.3 Ordre de les Transformacions 96

Pròleg

Després de 7 anys d'haver deixat abandonada la carrera d'Enginyeria Tècnica en Informàtica de Sistemes, m'he decidit a acabar les 4 assignatures que em quedaven per acabar, forçat pel fet que s'extingien aquests estudis degut a la implantació en el Procés Bolonya de l'Espai Europeu d'Ensenyament Superior (EEES).

Sempre he sigut una persona amb moltes inquietuds i he intentat formar-me en aquells sectors que les satisfessin. Així mentre estava acabant els estudis d'Arquitectura Tècnica, vaig començar a estudiar Electrònica, i mentre hi treballava vaig decidir començar els estudis d'Informàtica de Sistemes per omplir aquest buit de coneixements.

No obstant l'alt interès en aprendre moltes coses, mai he tingut la necessitat de finalitzar-los si no em podien aportar gran cosa més. Així els estudis d'Arquitectura Tècnica tot i acabar-los sense repetir cap assignatura, el projecte final de carrera no vaig presentar-lo fins 5 anys després quan s'extingia el pla vell. I ara passa el mateix amb els estudis d'Informàtica de Sistemes, que faig el treball final de carrera 7 anys després d'acabar-la.

No obstant aquest "abandonament" temporal, sempre he tingut clar que els treballs final de carrera no podien ser un simple tràmit. De la mateixa manera que quan la feina es torna repetitiva, trigo poc temps en deixar-la, no em veuria en cor de desenvolupar un treball final de carrera si no tingués aquell punt d'interès i repte que et manté viu. Així el projecte d'Arquitectura Tècnica fou un programa de càlcul d'estructures de barres en 3D, el projecte d'Electrònica el desenvolupament fou el d'un conjunt d'Autòmats Programables.

Actualment estic fent el Grau de Matemàtiques i volia mirar de fer un treball final de carrera que hi tingués relació. Així, la inspiració d'aquest treball ve del treball "Trencaclosques, jocs i matemàtiques" d'en David Arcos Gutierrez (Enginyeria de Telecomunicacions, Teoria de Jocs, 2010). Treball en que s'estudien els trencaclosques del Cub de Rubik de diferents dimensions finites i de dimensió genèrica (NxN).

En David Arcos és company en el Grau de Matemàtiques a la UB, on vaig interessar-me pel seu treball i sobre les possibilitats d'ampliar-se l'estudi a altres tipus de figures geomètriques, i com afectaria això a l'hora de trobar un algorisme de resolució genèric.

Com que no existeixen "cubs" de totes les dimensions i formes, es feia difícil poder realitzar l'estudi amb certa comoditat. Això em va fer pensar en crear un simulador que permetés crear "cubs" de diverses geometries i dimensions, i que servís d'interfície visual al sistema de resolució que es desenvolupés.

Al final, crec que he aconseguit l'objectiu pretès, de dissenyar i implementar el nucli d'aquest simulador, tot i que encara queden bastants aspectes molt interessants per completar la funcionalitat necessària per a que el simulador sigui una potent eina per a resoldre l'entrellat de l'estudi d'aquests trencaclosques.

Capítol 1

El Projecte

Aquest capítol descriu com es planteja el projecte, definit-ne els objectius que es pretenen aconseguir i descrivint la funcionalitat prevista. També es presentaran les eines i la tecnologia que s'utilitzarà per a dur-lo a terme.

1.1 Definició

El projecte '9Rubik' vol aconseguir crear un simulador en 3D (3 dimensions) de "Cubs de Rubik" en diverses geometries i dimensions. Aquest simulador empaquetat en una llibreria com un component ha de poder anar acoblat en qualsevol aplicació desenvolupada sobre la mateixa plataforma. A partir d'aquest, cada aplicació podrà oferir diferents funcionalitats utilitzant el simulador com a suport visual d'aquestes.

Les funcionalitats previstes del simulador queden resumides en el següent llistat:

- **Creació, càrrega i desat de "cubs"**
  S'oferirà missatgeria per a crear nous "cubs", així com desar-los i carregar-los al simulador.
- **Provocar el moviment de les peces**
  Hi haurà la missatgeria adequada per a realitzar tots els moviments de les peces del "cub".
- **Notificació dels moviments**
  Es faran notificacions de quan s'inicia un moviment i quin, quan s'atura, finalitza i/o es cancel·la; permetent a l'aplicació adaptar-s'hi.
- **Interactivitat mitjançant gestos**
  S'oferirà interactivitat gestual per a visualitzar el "cub" des de qualsevol posició, així com per a moure'n les peces.

El simulador es presentarà dins d'una aplicació mínimament funcional per mostrar-ne el funcionament bàsic. Aquesta aplicació permetrà seleccionar alguns "cubs" ja definits per a que l'usuari pugui, de forma interactiva, moure'n les peces, barrejar-les, així com re-iniciar-lo a l'estat inicial.

Altres aplicacions podrien oferir més funcionalitats, com per exemple:

- Crear interactivament nous "cubs" de diferents geometries i dimensions.
- Compartir els "cubs" amb altres usuaris.
- Implementar un sistema de joc i puntuacions sobre el 'Game Center' d'Apple.
- Afegir un sistema de registre de moviments com en els escacs.
- Afegir algorismes de resolució automàtics i/o guiats.

1.2 Objectius

Sense perdre de vista que l'objectiu principal del projecte és superar l'assignatura de 'Treball Final de Carrera' de l'Enginyeria Tècnica en Informàtica de Sistemes, s'ha de remarcar altres objectius importants que també s'assoliran:

- Entendre el funcionament dels "Cubs de Rubik".
- Aprendre a treballar amb la tecnologia OpenGL.
- Aprofundir en la programació amb el llenguatge Objective-C.
- Posar en pràctica l'aprenentatge de les llibreries de la plataforma iOS.
- Dur a terme un projecte de forma planificada.

1.3 Eines i Tecnologia

El projecte es desenvoluparà sobre la plataforma d'Apple per a dispositius mòbils anomenada iOS. Plataforma que permet desenvolupar aplicacions mòbils per a iPhone, iPod i iPad; des dels sistemes OSX d'Apple.

A falta de concretar-ne la disponibilitat, les eines que s'utilitzaran per a desenvolupar el projecte seran en quan a maquinari: un ordinador de sobretaula iMac amb 4Gb de RAM i sistema operatiu OSX 10.8, un iPad de 1a generació i un iPod Touch de 5a generació amb els certificats de desenvolupament corresponents.

Per a desenvolupar el simulador es disposa de: l'entorn de desenvolupament XCode d'Apple amb els respectius certificats de desenvolupament, el simulador 'iOS Simulator' per a les primeres proves de funcionament, i les aplicacions Instruments i 'OpenGL ES Performance Detective' per a controlar-ne el rendiment i les fuites de memòria.

Per a la redacció i presentació del projecte es treballarà amb: el llenguatge i processador de textos cientificotècnics LaTeX, l'aplicació de presentacions Keynote, i l'aplicació OmniGraffle per a crear gràfics i diagrames.

Les tecnologies en que es basarà el desenvolupament són bàsicament: les llibreries bàsiques per desenvolupar sobre iOS (Cocoa i UIKit), i la llibreria OpenGL per a produir gràfics en 3D.

Capítol 2

Pla de Treball

Aquest capítol presenta la planificació de com es durà a terme el projecte, mostrant-ne les fases del projecte, la seva viabilitat i identificant-ne les fases crítiques. També s'hi presenta el calendari de treball previst.

2.1 Fases del Projecte

En el llistat següent s'hi descriu cadascuna de les fases en que s'ha dividit el projecte. També s'hi indica la documentació que s'entregarà.

- **Preparació i planificació**: Aquesta fase té com a objectiu definir el projecte amb les seves funcionalitats. També s'hi identifiquen les fases i tasques a realitzar per dur a bon port el projecte, així com els riscos i el calendari de treball. S'entrega el document que esteu llegint.
- **Anàlisi**: L'objectiu d'aquesta fase és l'aprenentatge de les tecnologies necessàries (OpenGL, iOS), i l'anàlisi tècnic sobre com afrontar cadascuna de les tasques crítiques del projecte. S'entregarà un document detallat amb les solucions tècniques que s'utilitzaran en la fase de disseny i implementació.
- **Disseny**: Aquesta fase té per objectiu descriure els casos d'ús, el disseny de classes i les interfícies de l'aplicació. S'entregarà un document detallat amb els diagrames anteriors i la seves respectives descripcions, així com les captures de pantalla d'un prototip de l'aplicació.
- **Implementació**: L'objectiu de la implementació és aplicar les solucions tècniques de la fase d'Anàlisi sobre les classes de la fase de Disseny, per obtenir un simulador mínimament funcional. S'entregarà el codi font de l'aplicació i un fitxer executable per als dispositius prèviament autoritzats.
- **Presentació**: L'objectiu d'aquesta fase és lliurar i presentar el projecte mitjançant una memòria final que es basarà en el documents prèviament presentats, així com un full d'instruccions d'ús del simulador. També es farà una presentació tècnica en vídeo, juntament amb una demostració de funcionament del producte.

2.2 Viabilitat del Projecte

La viabilitat del projecte és bàsicament tècnica i de temps, i es pot desglossar en les següents tasques crítiques:

- **Creació del cub**: Una tasca poc crítica és la que ha de permetre crear cubs personalitzats, mitjançant missatges al simulador. Arribat el cas, aquesta funcionalitat podria substituir-se definint cubs des de fitxers.

- **Definició del cub**: Aquesta és la tasca més crítica ja que és necessari que cada "cub" quedi inequívocament definit per poder-lo simular.
- **Llenguatge de moviments**: La tasca de crear el llenguatge de moviments també és bastant crítica. La dificultat rau en definir un sistema de coordenades *global* i *local* que situï cada peça en el lloc i posició correcta en cada moviment.
- **Llenguatge de gestos**: Una tasca menys crítica és el llenguatge de gestos que permetrà a l'usuari manipular amb precisió el "cub". Cal aconseguir transformar els moviments dels dits en els girs adequats de les peces del "cub". Si no s'aconseguís, aquesta funcionalitat podria substituir-se mitjançant botons.

La resta de tasques que caldrà realitzar en el projecte són de pur tràmit i no es considera que puguin afectar-ne la viabilitat.

2.3 Calendari del Projecte

S'ha fet un calendari amb l'estimació de les tasques que caldrà fer i el temps que caldrà dedicar-hi.

Preparació i planificació	**8 dies**	**24/09/2012**	**01/10/2012**
Selecció del projecte	1 dia	24/09/2012	24/09/2012
Preparació del projecte	3 dies	25/09/2012	27/09/2012
Definició del projecte	1 dia	28/09/2012	28/09/2012
Planificació del projecte	1 dia	29/09/2012	29/09/2012
Redactat PAC1	2 dies	30/09/2012	01/10/2012
Entrega PAC1	Fita	01/10/2012	01/10/2012
Anàlisi	**18 dies**	**02/10/2012**	**22/10/2012**
Aprenentatge d'OpenGL	5 dies	02/10/2012	06/10/2012
Model de Negoci	8 dies	08/10/2012	16/10/2012
Interactivitat gestual	4 dies	17/10/2012	20/10/2012
Interfície d'usuari	1 dies	22/10/2012	22/10/2012
Disseny	**7 dies**	**23/10/2012**	**29/10/2012**
Disseny funcional	1 dies	23/10/2012	23/10/2012
Disseny de classes	1 dies	24/10/2012	24/10/2012
Disseny del prototipus	1 dies	25/10/2012	25/10/2012
Disseny de l'avaluació	1 dies	26/10/2012	26/10/2012
Redactat PAC2	3 dies	27/10/2012	29/10/2012
Entrega PAC2	Fita	29/10/2012	29/10/2012
Implementació	**36 dies**	**30/10/2012**	**10/12/2012**
Implementació del model	6 dies	30/10/2012	06/10/2012
Implementació dibuixat	4 dies	07/11/2012	10/11/2012
Implementació moviments	4 dies	12/11/2012	15/11/2012
Implementació selecció	5 dies	16/11/2012	21/11/2012
Implementació interacció gestual	6 dies	22/11/2012	28/11/2012
Implementació interfície aplicació	3 dies	29/11/2012	01/12/2012
Depuració i rendiment	2 dies	03/12/2012	04/12/2012
Publicació aplicació AppStore	2 dies	05/12/2012	06/12/2012
Preparació PAC3	4 dies	07/12/2012	10/12/2012
Entrega PAC3	Fita	10/12/2012	10/12/2012
Presentació	**9 dies**	**11/12/2012**	**07/01/2013**
Redactat memòria	12 dies	11/12/2012	23/12/2012
Presentació	8 dies	27/12/2012	05/01/2013
Entrega final	Fita	07/01/2013	07/01/2013

Figura 2.1: Calendari del Projecte

Capítol 3

Model de Negoci

Aquest capítol analitza en profunditat el funcionament dels “Cubs de Rubik”, el que s’anomena genèricament com el Model de Negoci.

En el model de negoci dels “Cubs de Rubik” s’hi estudiarà bàsicament la composició i funcionament d’aquests, per tal de poder definir-los de forma inequívoca i així poder crear-los i simular-los correctament. Això fa que sigui un capítol crític, ja que de les seves conclusions se’n derivaran les solucions de disseny que permetran desenvolupar el simulador.

3.1 Composició

El primer que s’ha de fer és identificar els elements que composen un “Cub de Rubik”.

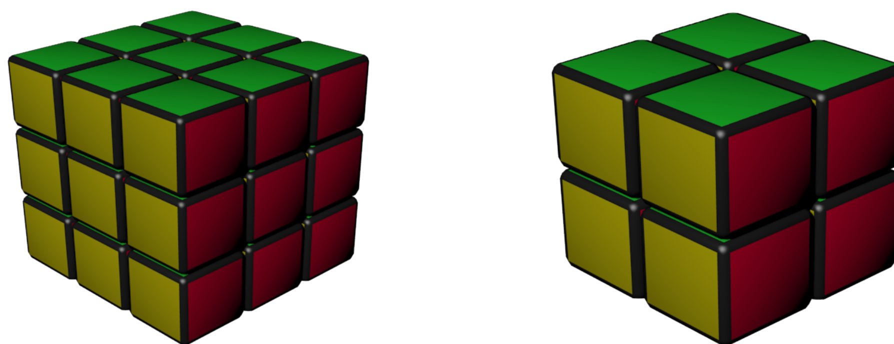

Figura 3.1: Cubs de Rubik clàssics: 3x3 — 2x2

3.1.1 La geometria

El primer que es pot identificar en un “Cub de Rubik” és la seva geometria o forma exterior. En els cubs clàssics —com bé diu el seu nom— tenen una geometria cúbica (hexàedre).

Podria semblar que aquesta geometria exterior és el que defineix els “Cubs de Rubik”, però no és així. Es pot veure en la Figura 3.2 i en qualsevol que hom pugui imaginar-se que es pot modificar l’aspecte exterior de qualsevol cub i aquest seguirà plenament funcional, ja que les peces poden seguir movent-se sense cap dificultat.

Per tant, es pot concloure que l’aspecte exterior és simplement un recurs visual per a que l’usuari

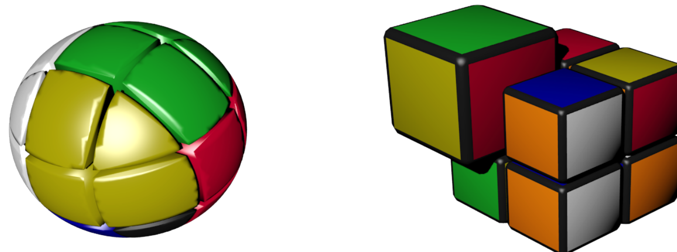

Figura 3.2: Esfera de Rubik — Cub deformat de Rubik

pugui resoldre amb més facilitat cada “Cub de Rubik”.

3.1.2 Els talls

Un altre element que s’identifica fàcilment són els talls que divideixen el “Cub de Rubik” en diferents peces. En la Figura 3.3 es pot veure que cada tall divideix completament el “Cub” formant un pla en l’espai.

Prenent com a base el cub de 2x2, que disposa de 3 plans de tall —perpendiculars entre ells—, se’n pot augmentar la complexitat afegint més talls paral·lels als existents. Així afegint-ne un més s’aconsegueix un cub de 3x3 i afegint-ne un altre un de 4x4, i així “fins a l’infinit i més enllà”.

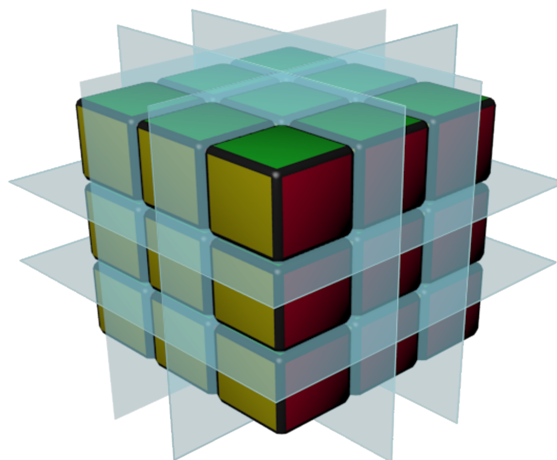

Figura 3.3: Cub de 3x3 amb talls

3.1.3 Les peces

Cada tall realitzat en el cub divideix la geometria en peces que seran les que es mouran i barrejaran, i que caldrà situar en la seva posició original per a resoldre el “Cub”.

Es pot observar en el cub original (Figura 3.3) que hi ha una peça en el centre que no és visible i per tant no afecta al “Cub”. Per tant els “Cubs” només han de tenir en compte les peces que limiten amb la superfície —sempre i quant no es treballi amb peces transparents.

Així un cub de 2x2x2 té 8 peces totes visibles, i un cub de 3x3x3 té 27 peces de les quals només 26 són visible. En el cas d’un cub de 4x4x4 tindria 64 peces, de les quals només 56 serien visibles deixant un petit cub de 2x2x2 al seu interior que no seria visible.

3.2 Funcionament

Un cop s'han identificat els elements que componen els "Cubs de Rubik" s'ha de mirar com intervenen dins el funcionament dels "Cubs" més enllà de l'obvietat.

Aquesta secció mirarà de treure l'entrellat de la relació que hi ha entre els seus elements i entre si mateixos, així com veure quines característiques han de tenir per a aconseguir definir un "Cub" plenament funcional.

3.2.1 Els plans

En la secció anterior s'han identificat els talls com els elements que dividien la geometria en les diferents peces. Si se simplifiquen els talls eliminant-ne els que són paral·lels i deixant-ne només un que el podem fer passar pel centre i anomenar-lo *pla* podem estudiar-ne les característiques que fan els "Cubs" viables.

Es veu clarament que els aquests plans són la base per als moviments de rotació de les les peces del "Cub". Per tant, com que aquests plans s'entrecreuen, d'alguna manera quan es realitza un moviment, les peces han de seguir un patró per a que la resta plans segueixin permetent més moviments.

Un pla únic

Començant l'estudi en el cas d'un únic pla, es pot veure en la Figura 3.4 com les peces poden girar de forma lliure entre ells sense cap restricció. A més, si es creen més talls paral·lels al pla el funcionament segueix essent el mateix.

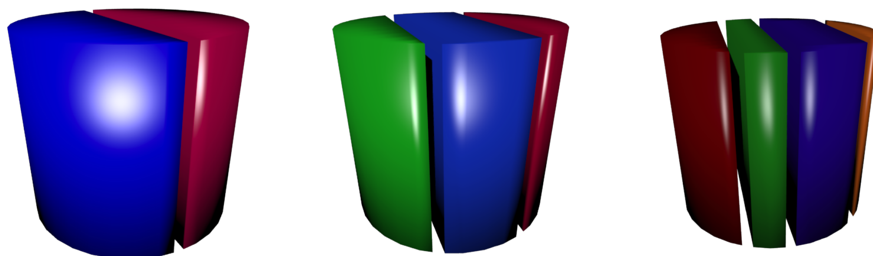

Figura 3.4: Figures amb un únic pla: 1 tall — 2 talls — 3 talls

Dos plans

Si s'amplia l'estudi i es fan servir dos plans es pot veure en la Figura 3.5 que hi ha un primer cas representat en la primera imatge on els plans són perpendiculars entre ells que faria el "Cub" viable i funcional; i un segon cas representat en la segona, tercera i quarta imatge on es veu que el cub no és funcional.

S'hi observa que si en el "cub" de la segona imatge es fa un gir sobre el pla vertical se n'obté la tercera imatge on clarament es veu que el moviment respecte a l'altre pla —l'inclinat a 60°— ha quedat invalidat. Si en canvi es fa un primer moviment sobre el pla inclinat se n'obté la quarta imatge i llavors el moviment del pla vertical és el que queda invalidat.

Generalitzant el segon cas amb un pla inclinat en la resta d'angles, és evident que només un pla perpendicular al vertical —com en el primer cas— permetria generar un "Cub" plenament funcional. En conclusió es pot dir que si tenim exactament dos plans, aquests han de ser perpendiculars entre ells.

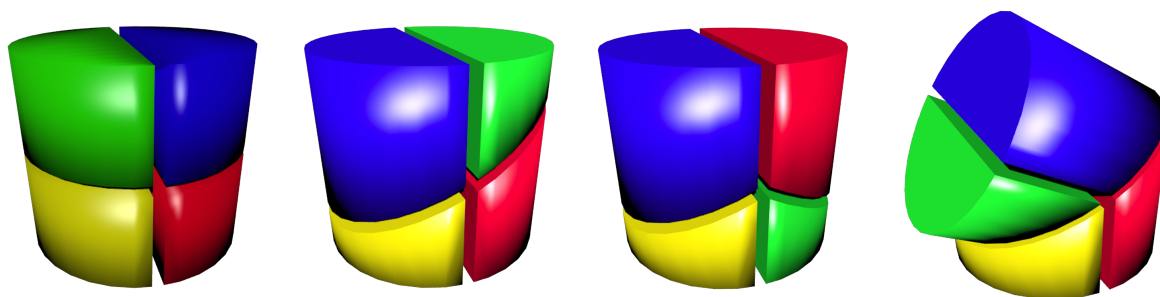

Figura 3.5: Dos plans: Perpendicular — 60º original — 60º gir vertical — 60º gir inclinat

Observant el "Cub" funcional es veu que els moviments que es poden fer amb les peces respecte cadascun dels dos plans són de 180º, o dit d'una altra manera: que per fer una volta completa cal fer dos moviments.

Plans no perpendiculars

S'ha vist que en el cas de tenir dos plans, aquests han de ser per força perpendiculars per a que el "Cub" funcioni, i coneixent el cas del Cub de Rubik clàssic que té tres plans que també són perpendiculars entre ells, es podria arribar a pensar que és una de les característiques per a poder tenir un "Cub" plenament funcional.

Figura 3.6: Talls no perpendiculars: Vista general — Vista zenital — Terra barrejada

Això no és cert, i l'exemple de la Figura 3.6 ho mostra amb una clara evidència. S'hi pot veure clarament un "Cub" plenament funcional amb tres plans que clarament no són perpendiculars entre ells. Els tres plans estan generats per revolució cada 60º ó 120º —depenent de com es miri— i amb angles entre ells de 60º.

Si es realitzen els moviments en algun dels plans s'hi observa que aquests han de ser de 180º, i també s'hi pot observar que si es giren juntament amb les peces la resta de plans, aquests al finalitzar un moviment vàlid ocupen el mateix lloc que ocupava un altre pla.

Clarament, el pla que genera la rotació manté la mateixa posició, i els altre dos plans al girar 180º intercanvien la posició, fent factible els seus respectius moviments.

Observant aquesta característica en els casos de plans perpendiculars entre ells, es pot observar en el cas de dos plans (Figura 3.7, que el gir —de 180º— sobre un pla provoca que l'altre pla —perpendicular al primer— també giri 180º i se situï en la mateixa posició. I en el cas del Cub de Rubik clàssic on es poden fer girs de 90º els dos plans al girar se situen en la mateixa posició que un l'altre, mantenint la plena funcionalitat dels "Cubs'.

Si s'amplia l'estudi en el casos de la Figura 3.8 amb 4 plans i 5 plans generats per revolució cada 45º

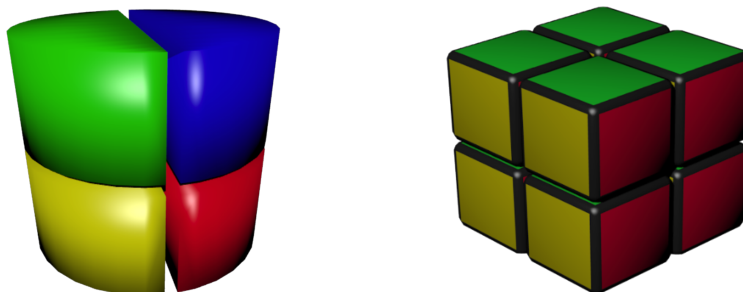

Figura 3.7: Talls perpendiculars: 2 plans — 3 plans

i 36º respectivament, s'hi pot observar com fent girs de 180º com en el cas anterior els "Cubs" també són funcionals, però en aquests casos es produeixen una variant que val la pena tenir en compte.

Figura 3.8: Talls no perpendicular: 3 plans — 4 plans — 5 plans

Es pot observar com al fer el gir de 180º respecte el pla horitzontal, el pla vertical de la imatge del mig queda en la seva mateixa posició i els altres dos s'intercanvien de forma semblant al cas de 3 plans. I en el cas de la imatge de la dreta també s'intercanvien els plans 2 a 2. Això vol dir que la resta de plans no cal que s'intercanviïn entre tots, sinó que es poden intercanviar en grups o fins i tot amb si mateix.

D'aquest anàlisi se'n pot concloure que una restricció o característica necessària per a poder tenir un "Cub" plenament funcional és el fet que els plans al girar se situïn en la mateixa posició on ja hi havia un altre pla. Així, el pla sobre el que es gira mantindrà la posició i cadascun dels altres plans al girar un cert nombre de graus hauran d'estar en el lloc d'un altre.

També se'n extrapola el fet que es girs han de ser en intervals divisors de 360º i que per cada pla que giri respecte d'un altre en generarà tants com intervals de gir es facin —tenint en compte que els plans perpendiculars girats 180º són el mateix pla.

Plans platònics

El cub és un dels 5 sòlids platònics, que són aquells poliedres, les cares dels quals són polígons regulars i que formen entre elles angles diedre iguals. N'hi ha cinc de diferents (Figura 3.9: el tetràedre, el cub o hexàedre, l'octàedre, el dodecàedre i l'icosàedre.

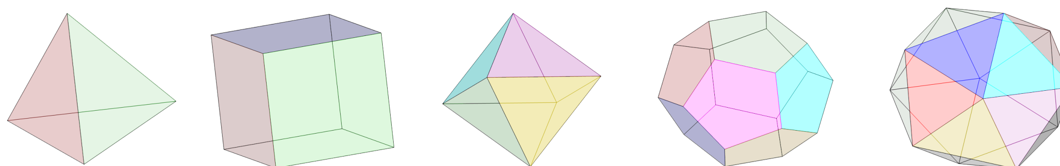

Figura 3.9: Sòlids Platònics: Tetràedre — Hexàedre — Octàedre — Dodecàedre — Icosàedre

Com que el Cub de Rubik té els plans paral·lels a les seves cares i com que els sòlids platònics tenen les cares amb angles diedres iguals, es plausible pensar que els plans generats a partir de les cares dels altres sòlids platònics també poden generar un “Cub” plenament funcional. En aquest punt s’analitzaran.

Tetràedre El tetràedre és un poliedre de 4 cares triangulars, que en la Figura 3.10 es pot veure clarament que és viable com a “Cub de Rubik”, ja que els seus plans respecte al de rotació es generen per revolució de 120º tal com es pot veure en la quarta imatge.

A la vegada i degut al fet que els plans són diedres sigui quin sigui el pla de rotació el resultat sempre és el mateix que es mostra en la darrera imatge de la Figura 3.10.

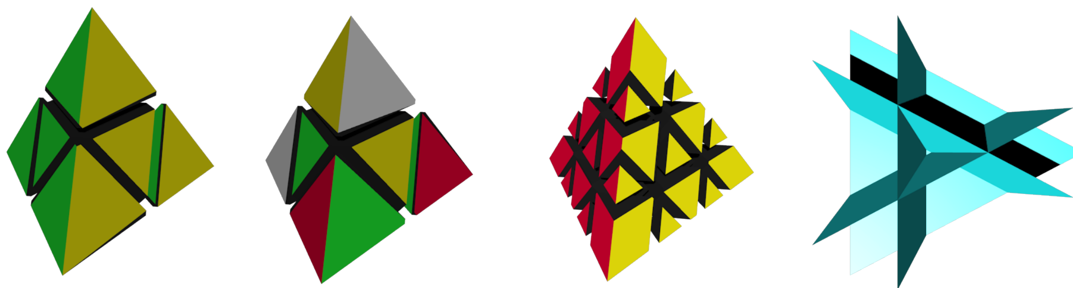

Figura 3.10: Tetraedre de Rubik: Original — Barrejat — de 3x3x3 — Plans de tall

Hexàedre L’hexàedre o cub ja s’ha analitzat amb anterioritat i sabem que les seves sis cares formen 3 plans perpendiculars entre ells i que respecte a un dels plans de rotació els altres dos es generen per revolució de 90º. Per tant els seus plans també formen un “Cub” vàlid.

Octàedre Si s’observa l’octàedre, si poden comptar 8 cares on dues a dues són paral·leles i per tant generen 4 plans. Aquests 4 plans prenen la mateixa posició que els del tetràedre que ja hem comprovat que generen un “Cub” plenament funcional.

Dodecàedre El dodecàedre té 12 cares, i com l’octàedre i el cub dues a dues paral·leles; per tant té 6 plans. S’hi veu fàcilment que fixant un dels plans s’aconsegueix que els altres 5 es generen revolucionant-los en intervals de 72º. Per tant els plans del dodecàedre també generen un “Cub” vàlid

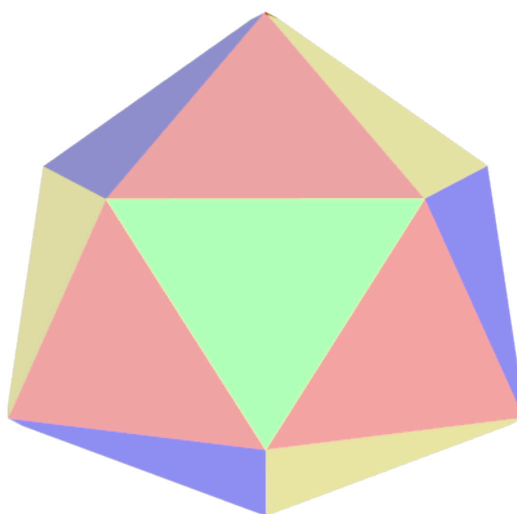

Figura 3.11: Icosaedre tallat per mig

Icosàedre El cas de l'icosàedre és més complex d'analitzar per la quantitat de cares que té. L'icosàedre té 20 cares, paral·leles dues a dues; per tant tenim un total de 10 plans. Si es tall pel mig amb un pla paral·lel a una de les cares, tal com es mostra en la Figura 3.11, s'hi poden observar els 10 plans reflectits en les cares visibles, ja que les cares paral·leles són a l'altra banda.

S'hi observa com els plans del mateix color —grocs, blaus i taronges— són generats per revolució en intervals de 120º respecte al pla de rotació —de color verd. Per tant, també es pot afirmar que els plans de l'icosaedre també generen un "Cub de Rubik" plenament funcional.

3.2.2 Els talls

Un cop els plans han quedat correctament definits, s'ha de veure com hi afecten els talls paral·lels a aquests plans. Si s'observar el Cub de Rubik clàssic es veu que tots els talls paral·lels tenen la mateixa separació i són simètrics respecte el centre de rotació. En aquest punt es veurà si aquestes característiques són comunes a tots els "Cubs" o no.

Plans perpendiculars

Si s'analitza el cas de dos plans on els girs són de 180º s'observar en la Figura 3.12 —on s'hi ha situat un tall desplaçat del centre—, que al fer el gir aquest tall es desplaça a la banda contrària del pla fent que ambdós talls siguin simètrics.

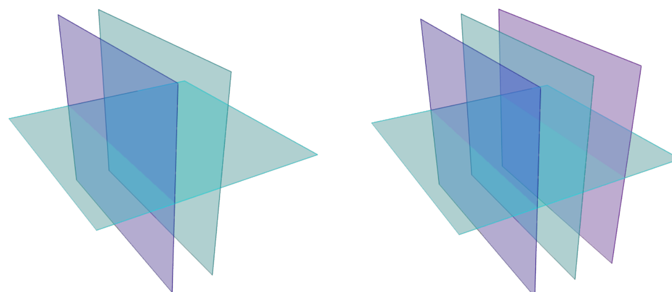

Figura 3.12: Gir de 180º en nivells de plans perpendiculars

Aquest cas fa concloure que en cas de tenir un pla perpendicular, si aquest gira 180º, els seus talls han de ser simètrics per a que el "Cub" sigui plenament funcional. Generalitzant es pot dir que si el nombre de passos per girar les peces 360º és parell, cada pla que gira haurà de tenir els talls de manera simètrica.

Plans no perpendiculars

A l'analitzar el cas de plans no perpendiculars que mostra la Figura 3.13, on els moviments també són de 180º, s'observar com el pla no se situa en la seva pròpia posició, però si que se situa en la posició d'un altre pla, per tant en condiciona la situació dels talls per a que el "Cub" segueixi sent plenament funcional.

En detall, la Figura 3.13 mostra en la primera imatge dos talls no simètrics afegits al pla '0' i com al realitzar un gir de 180º respecte del pla '1' aquests talls se situen en el pla '2' —tal com mostra la segona imatge— en la mateixa posició relativa que al pla '0' original. Fent el gir respecte del pla '2' passa el mateix —tal com mostra la tercera imatge—, però en aquest cas sobre el pla '1'.

En general s'ha de concloure que cada tall afegit a un pla s'ha de veure reflectit en la resta de plans que mantenen relació de moviment amb aquest primer, i en conseqüència recursivament amb la resta de

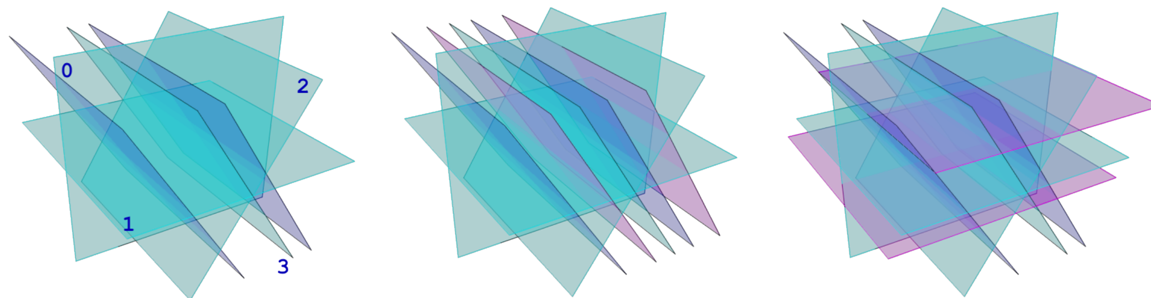

Figura 3.13: Girs de 60º o de 120º

plans que mantenen relació de moviment amb aquest altre, respecte el moviment en cadascun dels altres plans.

3.2.3 Els nivells

Cada cop que es vol moure una peça del "Cub', juntament amb aquesta se'n mouen d'altres amb les que hi ha una restricció de moviment. Aquestes peces que es mouen conjuntament es diu que són en el mateix nivell. Un nivell queda definit per l'espai que hi ha entre dos talls consecutius, i per l'espai dels talls dels extrems enllà.

Es pot observar que n talls paral·lels sobre un pla generen una partició de $n + 1$ nivells amb peces que s'han de moure conjuntament. Així, el cub de 2x2x2 té tres plans de tall que generen un total de 6 nivells de gir. En el cas del cub de 3x3x3 té 3 plans de tall no paral·lels amb 2 talls paral·lels per a cadascun que generen un total de 9 nivells de gir.

3.2.4 Les peces

Les peces del "Cub de Rubik" són els elements visibles que hom veu al manipular-lo i d'alguna manera han d'estar relacionades amb la resta d'elements que s'han estudiat fins ara. Aquest punt vol veure quina relació s'hi estableix.

La posició

Cada peça del "Cub" pertany a un, i només un, dels nivells de cada pla. Per saber en quin nivell d'un pla pertany una peça —suposant que la seva geometria hi estigui restringida—, s'ha de calcular a quina distància és qualsevol punt de la seva geometria respecte el pla. Un cop coneguda aquesta distància es pot comparar respecte a la distància de cadascun dels talls i saber a quin nivell pertany.

3.2.5 La solució

No es pot deixar en l'oblit un element molt important en el funcionament d'un "Cub de Rubik", i aquest no és sinó la solució d'aquest.

Es pot pensar que per a que un "Cub" estigui solucionat, simplement s'ha de col·locar cadascuna de les peces en la seva posició original. Però això no és del tot cert. I és el que es mirarà d'aprofundir en aquest punt.

El punt de vista

Un primer concepte evident és el fet que la posició original no és el que defineix un "Cub" solucionat, sinó que ho és la posició relativa respecte a la resta de peces. Així, clarament, si en un "Cub" es giren tots els nivells un mateix nombre de graus, la orientació del cub o el seu punt de vista ha canviat però la posició relativa de totes les peces segueix essent la mateixa, i per tant el "Cub" també està resolt.

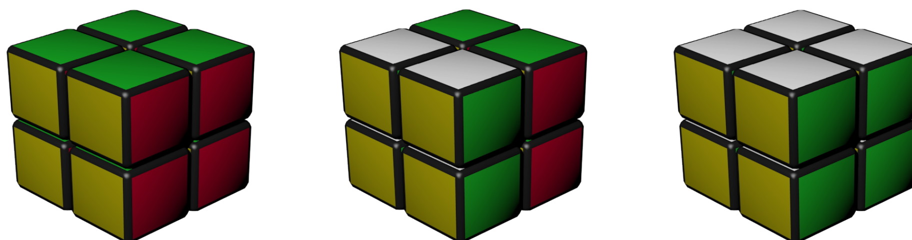

Figura 3.14: Posició original — Gir nivell 1 — Gir nivell 0

La posició

Ja s'ha comentat que la geometria exterior no condicionava el funcionament dels "Cubs" i que simplement eren una ajuda per la seva resolució. No obstant, cal remarcar que pot existir el cas que dues peces d'un "Cub" tinguin la mateixa forma exterior i per tant puguin ser intercanviables sense que se'n vegi la diferència. En la Figura 3.15 es poden diferenciar clarament que només hi ha tres peces amb formes diferents: la verda quadrada, la vermella rectangular i la blava triangular; i per tant poden intercanviar-se entre ells.

Figura 3.15: Esfera blanca — Esfera amb colors — Esfera amb textura

La orientació

Seguint amb la Figura 3.15, la forma exterior de la peça també pot fer que una mateixa peça pugui adoptar diferents orientacions en la mateixa posició. Per exemple la peça amb forma triangular —de color blau— podria adoptar 3 posicions sense que se'n veiés la diferència, la peça vermella 2 posicions diferents i la verda 4 de diferents.

La textura

Seguint l'anàlisi de la Figura 3.15, s'observa en la tercera imatge la textura de la peça és un element important per a que peces amb la mateixa forma puguin situar-se en diferents posicions dins del "Cub" o que una mateixa peça pugui col·locar-se en diferents orientacions.

Per exemple, en la segona imatge es pot veure que el fet que dues peces d'igual forma tinguin colors diferents ja impedeix que puguin intercanviar-se, ja que visualment es veuria que no encaixen. I si a la

vegada la peça disposa d'una textura que s'ha de fer quadrar amb la resta de peces que l'envolten, també impedeix que pugui col·locar-se en una altra orientació.

3.3 Manipulació

Aquesta secció vol analitzar com es realitza la manipulació del "Cub de Rubik", així com queden definides per a poder-les traslladar a una futura implementació.

3.3.1 Movent el "Cub"

Quan es té un "Cub" entre les mans el primer que es fa és moure'l en totes les posicions per veure'n les diferents peces.

Des d'un punt de vista tècnic la millor forma de traslladar aquest moviment no és movent el "Cub" en si, que implicaria moure cadascun dels plans, i de les peces, sinó moure el punt de vista des d'on se l'observa. Així, si s'observa el "Cub" des d'una vista zenital —des de sobre— i el girem 90° respecte l'eix Y, seria el mateix que si es mirés des de la dreta.

3.3.2 Movent les peces

L'altre acció que es realitza amb un "Cub" és moure'n les peces. Això fa que aquestes es barregin i s'hagi de solucionar ficant-les totes al seu lloc. Aquest moviment s'haurà de fer per a cadascuna de les peces que es mouen en cada cas, tenint en compte que el moviment és faci en els intervals exactes definits per a que el cub sigui funcional

De l'anàlisi del funcionament dels "Cubs de Rubik" se n'ha conclòs que els nivells són els elements que restringeixen els moviments de les peces del "Cub". Per tant el moviment no es realitza sobre les peces en sí, sinó sobre un dels nivells d'un dels plans.

Capítol 4

Tècnica i Tecnologia

Aquest capítol fa un breu resum de la tecnologia que s'utilitzarà en aquest projecte, això com la tècnica matemàtica que s'aplicarà per a solucionar diversos aspectes de funcionament.

4.1 Objective-C

El llenguatge Objective-C és una extensió del llenguatge C, dissenyat per permetre la programació orientada a objectes inspirant-se en la sintaxi del llenguatge Smalltalk. Dissenyat per en Brad Cox l'any 1980 i popularitzat el 1988 per l'Steve Jobs a l'adoptar-lo com a llenguatge de programació del sistema operatiu orientat a objecte NeXTSTEP que fou el projecte que va engegar en l'empresa NeXT Computer després que el fessin fora d'Apple.

L'any 1992 va ser alliberat sota llicència GPL i s'ha popularitzat encara més quan l'Steve Jobs va tornar a Apple i va basar el nou sistema operatiu Mac OS X d'Apple en el seu desenvolupament de NeXTSTEP.

4.1.1 La nomenclatura

Una de les característiques que es diferencia a altres llenguatges orientats a objectes fa referència al llenguatge utilitzat. Així mentre que en la majoria d'aquests llenguatge es criden o executen mètodes de l'objecte o de la classe, en Objective-C es diu que s'envien missatges a l'objecte o a la classe. Així mateix, l'objecte o classe que rep el missatge s'anomena "receptor" mentre que l'"emissor" seria l'objecte des d'on s'envia el missatge.

Això no és quelcom casual i semàntic, sinó pel fet que mentre que en C++ el mètode cridat queda enllaçat en el seu objecte pel compilador, en Objective-C el compilador simplement envia el nom del missatge que es resol en temps d'execució. Això fa que no hi hagi seguretat que un objecte receptor respongui a un missatge si no el té implementat i l'ignori retornant un punter nul.

4.1.2 La codificació

A l'hora d'escriure codi les característiques més importants, i visuals, fan referència principalment a com es defineixen els mètodes i com s'hi envien els missatges.

```
// Definició i crida en C++ o Java
int methodName(type1 param1, type2 param2, ...);
object.methodName(param1, param2, extra, params);
```

```
// Definició i enviament de missatge amb Objective-C
- (int) messageSendedWithParam1:(type1)param1 andParam2:(type2)param2, ...
[receiver messageSendedWithParam1:param1 andParam2:param2, extra, params];
```

Mirant el codi anterior s'observar una primera diferència evident i és el fet que la definició del mètode no tanca els paràmetre entre parèntesi, sinó que estan inclosos dins els nom del mètode —o el missatge enviat— precedit del caràcter de dos punts (':'), fent que el mateix nom del missatge pugui descriure cadascun dels paràmetres. Tot i que no és evident en Objective-C, el nom del mètode o missatge no queda definit pels tipus de paràmetre com si que es fa amb C++ i Java i per tant no es poden enviar missatges iguals amb paràmetres de diferents tipus. El nombre de paràmetres si que hi queda definit degut a la utilització dels dos punts (':'). Així, seguint l'exemple anterior, el receptor (`receiver`) rebrà el missatge `messageSendedWithParam1:andParam2:` i així és com s'identifiquen inequívocament —sense espais i tipus de paràmetres. Aquestes característiques en la definició dels missatges fa que el propi nom del mètode sigui ja una descripció del que fa, amb quins paràmetres o fa, i quin valor retorna, fent gairebé redundant afegir-hi comentaris.

Quan s'envia un missatge a diferència de C++, és fa entre dos claudàtors on el primer paràmetre és l'objecte receptor del missatge i la segona part el missatge enviat amb els paràmetres necessaris. També com en C++ es poden afegir paràmetres extres afegint-hi els tres punts ('...'). Aquesta forma d'enviar els missatges fa que no es puguin concatenar crides amb un simple punt, però alhora evita afegir-hi parèntesi (les claudàtors ja fan el fet) per diferenciar qui rebrà el missatge.

Una altra característica que es veu és que tots els tipus dels paràmetres i de l'objecte retornat sempre queden definits entre parèntesi. També es pot observar en la definició del mètode el signe menys ('`-`') al seu davant. Aquest signe serveix per diferenciar els mètodes d'objecte (amb signe menys '`-`') dels mètodes de classe (amb signe més '`+`').

4.1.3 Gestió d'instàncies

Un altre aspecte important de l'Objective-C és com els objectes gestionen els objectes instanciats. Bàsicament aquesta gestió es fa sobre la base d'unes convencions bàsiques en la definició dels mètodes de les classes.

El objectes en Objective-C contenen un comptador que indica quantes variables la referencien, per tant cada cop que un objecte vol tenir accés a un objecte ha d'indicar-ho enviant-li el missatge `retain`, i quan ja no el vol utilitzar més també amb el missatge `release`. Això fa que el comptador de l'objecte accessible mitjançant el mètode `retainCount` sàpiga quan es pot esborrar. Així, quan el comptador arriba a zero l'objecte crida el mètode `dealloc` que s'encarrega d'alliberar la memòria prèviament reservada.

Cal fer esment també del missatge `autorelease` que és una delegació de l'alliberament a un objecte de recol·lecció de brossa (`NSAutoreleasePool`) que s'encarrega d'alliberar els objectes que han rebut aquell missatge a posteriori. Per defecte l'aplicació en té una que es buida a cada cicle, però el programador en pot afegir altres dins de funcions per alliberar memòria de forma més organitzada.

Les convencions en els mètodes ens diuen que els mètodes que comencen per `alloc` i `copy` retornen un objecte nou, és a dir que ja se li ha incrementat el comptador. Els mètodes `alloc` s'encarreguen de crear un objecte per primera vegada reservant l'espai necessari per a les seves variables inicialitzades a zero, i els mètodes `copy` en creen un de nou copiant-ne el contingut.

Com que els mètodes `alloc` acostumen a estar definits per defecte s'utilitzen els mètodes que comencen per `init` per inicialitzar-ne el contingut depenent del seu nom i dels paràmetres que se li passen. Els mètodes init també són els encarregats de validar el nou objecte abans de retornar-lo o esborrar-lo de la memòria si no és viable. Cal destacar que els mètodes init poden substituir l'objecte a inicialitzar per un altre o simplement esborrar-lo, i per tant el programador ha de comprovar el què retorna.

La resta de mètodes per convenció retornen objectes als quals se'ls ha enviat un missatge de `autorelease` i per tant si es volen utilitzar fora del mètode cal incrementar-ne el comptador enviant el missatge `retain`.

4.1.4 Altres característiques

Altres característiques interessants a enumerar són l'existència de protocols. Existeixen els protocols formals que són bàsicament com les interfícies de Java i els protocols informals que també s'anomenen Categories.

Els formals es defineixen amb la directiva `@protocol`, i bàsicament és la definició d'una sèrie de mètodes opcionals o obligatoris que han d'implementar les classes que adoptin aquest protocol. En canvi els informals o categories es defineixen amb la directiva `@category`, i permeten estendre la funcionalitat d'una classe afegint-hi noves variables, i creant nous mètodes o modificant el funcionament dels existents. Es poden crear tantes categories com es vulgui.

Una altra característica peculiar és l'anomenada "Posing" que permet substituir durant l'execució una classe per una altra. És a dir, que tots els objectes d'una classe passen a ser considerats d'una altra classe i per tant serà l'encarregada de respondre als missatges.

En la versió 2.0, el llenguatge ha afegit noves característiques que l'acosten més als llenguatges d'alt nivell moderns, que simplifiquen el codi per fer coses senzilles però que obliguen al programador a memoritzar un munt de nous conceptes amb múltiples variants que sovint dificulten moltes tasques.

4.2 Cocoa

Cocoa és la API nativa i orientada a objectes d'Apple per al sistema operatiu Mac OS X, i Cocoa Touch és la versió de la API per als dispositius mòbils. Aquesta API és l'evolució de les APIs originals de NeXTSTEP, per això totes les seves classes comencen amb l'acrònim `NS`.

4.2.1 Prefixos

S'ha vist que les classes de les llibreries bàsiques de Cocoa comencen amb el prefix `NS` (de NeXTSTEP). Això és una convenció per a diferenciar a quina llibreria pertany cada classe i evitar que hi hagin classes de llibreries diferents amb el mateixos noms.

Observant una mica les diverses llibreries que proporciona Apple, hi veiem la relació amb els prefixos utilitzats. El prefix `AV` s'utilitza per a les classes d'Àudio i Vídeo, la `UI` per a la interfície d'usuari sobre iOS. També n'hi ha de més especialitzades com: `AB` per a les llibreries de l'Agenda (Address Book), `GK` per a les de joc (Game Kit), `EK` per als esdeveniments (Event Kit), `AD` per als anuncis, `MK` per als mapes (Map Kit), `TW` per al Twitter, entre moltes d'altres.

4.2.2 Model–Vista–Controlador

La programació amb Cocoa acostuma a seguir el patró de disseny "Model–Vista–Controlador" (MVC). Bàsicament el MVC es basa en tenir tots els objectes en una partició basada en tres sub-sistemes: el model, la vista i el controlador.

El conjunt Model conté les classes que representen les dades amb que treballa l'aplicació així com el seu funcionament inherent. Aquest conjunt ha de ser autosuficient i independent de la resta de l'aplicació.

El conjunt Vista conté les classes que mostren la informació i interactuen amb l'usuari. Aquestes classes estan fortament lligades amb el dispositiu que manipula l'usuari, ja sigui una impressora, una pantalla o un sistema tàctil.

El conjunt Controlador és l'encarregat d'unir els altres dos conjunts, fent que les dades es mostrin a la vista de forma adequada tractant-les prèviament si cal, i aplicar els canvis sobre el Model quan l'usuari interactua sobre la vista.

4.3 XCode i Instruments

L'XCode és l'entorn de desenvolupament per a fer aplicacions per a OS X i iOS. Conté el compilador gcc, així com l'editor d'interfícies gràfiques: Interface Builder. També molt important incorpora l'extensa i detallada documentació per a desenvolupar sobre Apple incloent-hi manuals molt complets que introdueix al programador en els diverses tècniques de disseny i programació d'Apple.

Dins l'XCode també hi ha l'aplicació Instruments que és una aplicació per visualitzar i analitzar el rendiment de les aplicacions. Aquesta aplicació està feta sobre la framework DTrace d'OpenSolaris. Existeixen eines per analitzar la memòria, l'activitat del monitor, el consum d'energia, la xarxa, les llibreries OpenGL, entre altres.

4.4 OpenGL

L'OpenGL (Open Graphics Library) com el seu nom diu és una llibreria gràfica oberta, multi-llenguatge i multi-plataforma. Fou desenvolupada inicialment per Silicon Graphics per a ser utilitzada en les seves estacions gràfiques de CAD, realitat virtual, etc. Amb la proliferació de targetes gràfiques de la competència que començaven a introduir un nou estàndard, van decidir convertir el seu estàndard en obert anomenant-lo OpenGL. Aquesta llibreria s'ha transformat en una especificació que poden seguir fabricants de targetes gràfiques que la vulguin implementar.

Aquesta especificació es basa en una interfície que comunica l'aplicació amb la targeta gràfica per aconseguir el mateix resultat sigui quin sigui el fabricant i la plataforma sobre el que s'executa.

Les instruccions es basen en representar punts, línies i polígons; tant en 2D com en 3D, indicant com cal dibuixar-les

4.5 Geometria

En aquesta secció es descriu en detall la tècnica matemàtica relacionada amb la geometria que s'utilitzarà a l'hora de programar el simulador.

4.5.1 Equació General del Pla

Per treballar amb plans cal definir-los, i això es pot fer per mitjà de l'equació general o cartesiana del pla. Aquesta equació té la forma $Ax + By + Cz + D = 0$ on el vector (A, B, C) és la normal del pla i la D és la distància d'aquest al pla paral·lel que passa pel punt zero "(0,0,0)". En la Figura 4.1 es mostra un pla amb la seva normal.

L'equació general s'obté a partir d'un punt del pla, en el cas present els plans passen pel centre de coordenades: $p_0 = (x_0, y_0, z_0)$; i dos vectors d'aquest: $u = (u_1, u_2, u_3)$ i $v = (v_1, v_2, v_3)$. Així, l'equació s'obté desenvolupant el següent determinant igualat a zero.

$$\begin{vmatrix} p - p_0 \\ u \\ v \end{vmatrix} = 0 \Rightarrow \begin{vmatrix} x - x_0 & y - y_0 & z - z_0 \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = 0 \Rightarrow Ax + By + Cz + D = 0$$

4.5.2 Posició respecte el Pla

Per saber la posició relativa d'un punt respecte el pla només s'ha de substituir el punt a l'equació general i observar-ne el resultat. Si el resultat és zero, el punt pertany al pla. Si el resultat és major de zero el punt està a un costat del pla, i si és menor de zero a l'altre costat del pla.

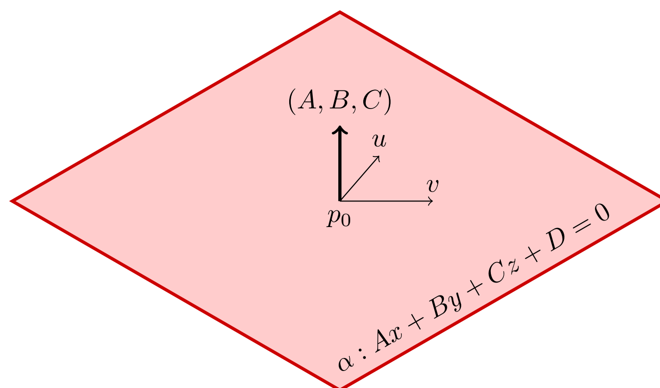

Figura 4.1: Pla $\alpha : Ax + By + Cz + D = 0$

Per exemple, suposant el pla $0x+0y+z = z = 0$ en que passa pels eixos x i y, els punts $p_0 = (3, 3, 0)$, $p_1 = (1, 1, 2)$ i $p_2 = (1, 2, -1)$ respecte a aquest pla $z = 0$, queden a:

$$\begin{cases} p_0 = (3,3,0): & z = 1(0) = 0 & \Rightarrow \text{pertany al pla} \\ p_1 = (1,1,2): & z = 1(2) = 2 & \Rightarrow \text{és sobre el pla} \\ p_2 = (1,2,-1): & z = 1(-1) = -1 & \Rightarrow \text{és sota el pla} \end{cases}$$

La Figura 4.2 mostra el pla i la posició dels punts respecte a la superfície del pla.

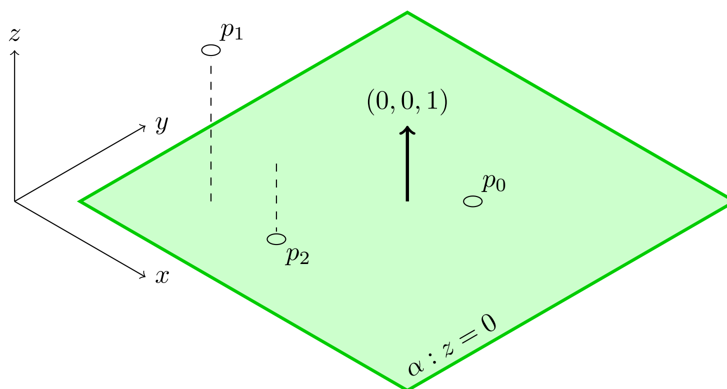

Figura 4.2: Pla $\alpha : z = 0$ i punts: p_0, p_1 i p_2

4.5.3 Interseccións

Treballant amb elements geomètrics és molt comú haver de trobar el resultat de la intersecció entre plans, o entre plans i rectes.

Intersecció Pla–Pla

La intersecció entre dos plans dona una línia que és comuna a aquests dos plans definida per un vector director de la recta si ambdós plans passen pel punt zero $(0, 0, 0)$.

A efectes matemàtics aquest vector director és la normal que defineix un pla perpendicular als dos plans anteriors. I aquest vector es pot calcular amb un simple producte vectorial de les normals dels primers dos plans. El producte vectorial és el desenvolupament de la següent matriu amb les normals

dels plans (u i v). Així:

$$\begin{vmatrix} x & y & z \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = Ax + By + Cz = (A, B, C)$$

La Figura 4.3 mostra la intersecció entre dos plans, i la intersecció entre un pla i una recta.

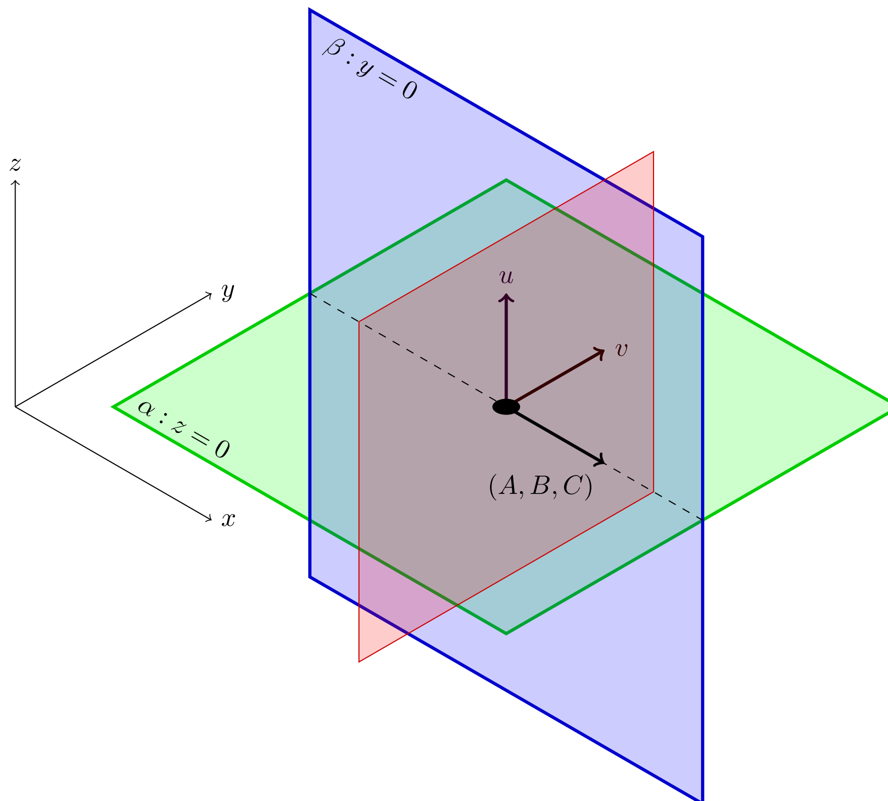

Figura 4.3: Plans α i β: Interseccions pla–pla i recta–pla

Intersecció Recta–Pla

Tot i que per al projecte no s'ha hagut d'utilitzar la intersecció entre una recta i un pla, per trobar quines peces se seleccionaven amb els dits, es interessant veure com es produeix aquesta intersecció. S'utilitzarà com a referència la Figura 4.3.

Ja s'ha vist que una recta és la intersecció entre dos plans, per tant la intersecció entre un pla i una recta, és el mateix que la intersecció entre tres plans. Però s'han de trobar aquests dos plans.

Per definir els dos plans que generen la recta simplement s'ha d'agafar un punt de la recta i el vector de la recta i combinar-los conjuntament amb dos vectors diferents, un per cada pla. El del primer pla és pot agafar fàcilment intercanviant dos coordenades amb valors diferents i el signe d'una d'elles. Per al segon pla simplement fent el producte vectorial dels dos vectors del primer pla ja se n'obté un de diferent.

Un cop es tenen les tres equacions dels plans simplement se n'ha de resoldre el sistema d'equacions de tres incògnites, que donarà un punt en l'espai.

4.5.4 Matriu de Rotació

Per girar els elements en l'espai respecte d'un eix —que pot ser la normal que defineix un pla— cal multiplicar cada punt de l'espai per una matriu de rotació definida a partir de la direcció de l'eix i l'angle

de rotació. Tot i que l'OpenGL disposa de mètodes per calcular una matriu de rotació, no se'n pot obtenir els valors ja que queden desats en la matriu de transformació dins del propi OpenGL. La llibreria `GLKit` de l'iOS, també ofereix aquesta funcionalitat, però només està disponible a partir de la versió iOS 5. Per tant, si es vol generar rotacions sobre certs elements, s'ha de poder generar aquesta matriu.

S'ha de tenir en compte que la matriu de rotació és diferent depenent de si es multiplica amb el vector a girar a la dreta o a l'esquerra, i tenint en consideració què s'entén per un gir positiu o negatiu. Aquí s'ha muntat la matriu de rotació tal qual la utilitza l'OpenGL, per a poder-li enviar com a transformació i l'apliqui directament; i per tant el vector a girar serà horitzontal i multiplicarà amb la matriu a la seva dreta.

La matriu de rotació s'obté a partir del vector vertical de l'eix de rotació que anomenarem u i de l'angle de rotació que anomenarem α.

$$R = u \cdot u^t + cos\alpha(I - u \cdot u^t) + sin\alpha \begin{pmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{pmatrix}$$

4.5.5 La Càmera en l'Escena

La càmera que ha de definir el punt de vista de l'escena esta definida per un vector de posició i un angle de gir respecte la vertical. Això dificulta una mica les operacions necessàries (desplaçaments i rotacions) que cal realitzar amb l'OpenGL per aconseguir el resultat desitjat. Tenint en compte que l'OpenGL no desplaça la càmera, sinó que desplaça l'escena per a que es vegi sempre zenitalment des del punt $(0, 0, 0)$.

En la Figura 4.4 veiem com una escena normal amb l'objecte en el seu centre i la càmera al seu voltant ha de desplaçar-se per a aconseguir que la càmera estigui en el punt $(0, 0, 0)$ mirant zenitalment amb la vertical sobre l'eix Y.

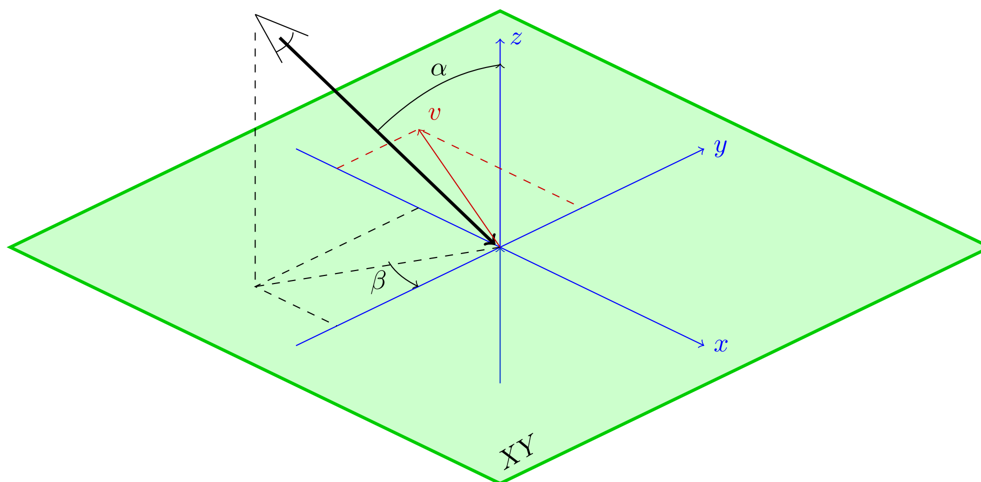

Figura 4.4: Posició de la càmera i rotacions en l'escena

Considerant que el vector $c = (c_x, c_y, c_z)$ indica la posició de la càmera. El primer que cal fer és desplaçar tota l'escena per a que la càmera se situï en el punt $(0, 0, 0)$ i això és fa fàcilment fent una translació inversa a la posició d'aquesta $-c$ unitats.

Un cop tenim això s'ha de girar l'escena perquè la càmera miri zenitalment. Per trobar l'angle (α) de rotació calculem arc-tangent entre la projecció de la posició de la càmera sobre el pla XY i l'eix Z. I el vector de rotació (v) definit per un vector perpendicular contingut en el pla XY.

$$\alpha = \arctan\left(\frac{\sqrt{c_x^2 + c_y^2}}{c_z}\right), \qquad v = (c_y, -c_x, 0)$$

Per acabar amb l'escena sota la càmera només queda girar-la respecte a l'eix Z per a alinear la vertical de la càmera amb l'eix Y de l'escena i afegir-hi l'angle de gir d'aquesta. Per trobar aquest angle simplement trobem l'angle (β) que forma la posició de la càmera respecte a l'eix Y i li sumem la rotació (γ) definida.

$$\beta = \arctan\left(\frac{c_x}{-c_y}\right)$$

Només queda aplicar la projecció ortogonal o en perspectiva amb el zoom adient. Sobre l'OpenGL aquests moviments s'han d'executar en ordre invers degut a que es multiplica la matriu de rotació per la dreta.

4.5.6 Moviment de la Càmera

El primer moviment a considerar de la càmera és la rotació d'aquesta sobre si mateix —com si es girés el cap. Per fer això no hi ha cap més dificultat que incrementar o decrementar la variable de rotació que la defineix.

Més complex és moure la càmera al voltant del "Cub de Rubik" per simular que s'està movent el "Cub". Per fer això cal girar la posició de la càmera al voltant d'aquest, però més important que això, és establir sobre quin vector es gira la càmera, quin angle i en quin sentit; depenent del desplaçament que s'hagi fet amb els dits sobre la pantalla.

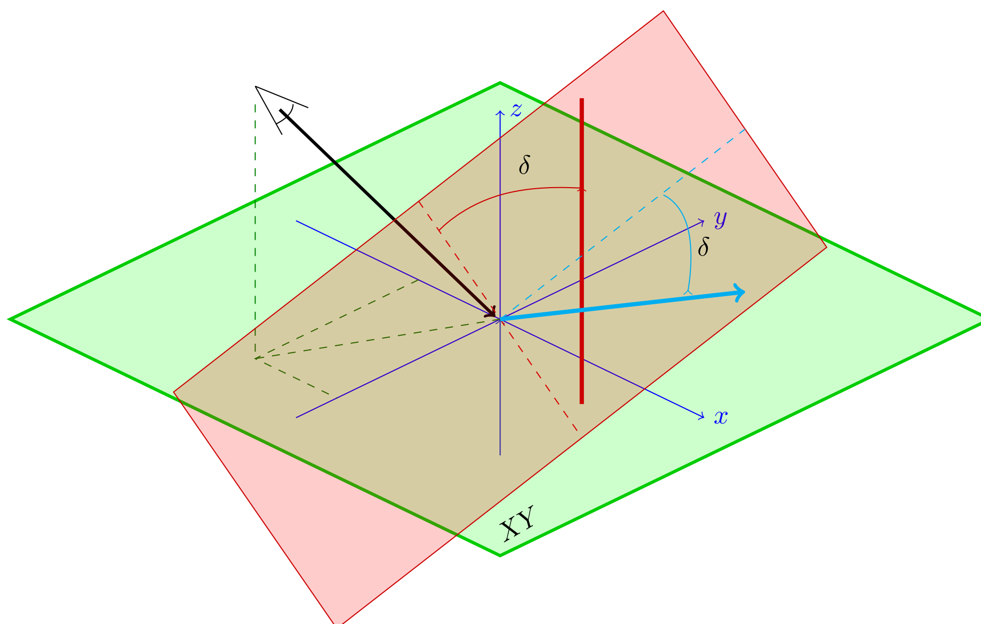

Figura 4.5: Eix de rotació de la càmera

Si s'observa la Figura 4.5 s'hi pot veure el punt de vista amb un ull de color negre i la pantalla de projecció de color vermell que talla amb el pla XY representada amb una línia discontínua també vermella. Quan l'usuari traça un moviment amb el dit sobre la pantalla de projecció es genera un vector gruixut de color vermell a partir del punt inicial i el final, el qual forma un angle (δ) respecte la horitzontal.

Amb aquest vector, la mida de la pantalla i la posició de la càmera es calcula el vector i angle de rotació de la càmera. L'angle de rotació s'obté de relacionar la longitud del vector respecte la màxima longitud de la pantalla, sent aquesta última equivalent a una rotació de 180º. És a dir que si es desplaça d'un extrem a l'altre de la pantalla es gira la càmera 180º.

Per trobar el vector de gir és una mica més complex. El primer que cal fer és trobar la intersecció entre el pla XY i la pantalla (vist a la Secció 4.5.3). Aquest vector girat 90º (discontinu de color cian) més l'angle δ (de color cian), més el gir propi de la càmera respecte l'eix representat per la posició de la

càmera ens dona el vector de rotació de la càmera. Ja només cal girar la càmera multiplicant el vector de la seva posició per la matriu de rotació calculada pel vector i angle de rotació (vist a la Secció 4.5.4).

Un canvi a considerar per millorar la interacció amb l'usuari és traduir l'angle de gir respecte la vertical per un vector que n'indiqui la direcció vertical de la pantalla, i també rotar-lo juntament amb la posició de la càmera. Això evitaria l'efecte de gir de la càmera al passar pels pols en el seu gir.

4.5.7 Sentit del Moviment

Per decidir el moviment de les peces, o per ser més curosos, el moviment d'un nivell de peces, es fa anant seleccionant peces i quan aquestes només tenen un únic nivell en comú, moure aquest nivell. La quantitat de moviment sempre serà l'interval que permeti el seu pla.

Tanmateix, saber en quin sentit s'han de moure les peces —si en sentit horari o trigonomètric— no és tant senzill. En aquest punt se'n descriurà l'algorisme i les matemàtiques.

Agafant els dues primeres cares de les dues primeres peces seleccionades, podem obtenir-ne dues coordenades (u i v). Fent un producte vectorial d'aquests dos vectors (conservant l'ordre de selecció) se n'obté un vector perpendicular al pla generat per aquests dos. Ara només cal comparar aquest vector unitari, amb la normal del pla de rotació. Si l'angle entre ells és menor de 90º el sentit de gir es considerarà positiu (trigonomètric) i si és major de 90º es considerarà negatiu (horari).

4.6 OpenGL

4.6.1 Aplicar la Textura

Per poder utilitzar textures en OpenGL cal enviar-li les dades però amb algunes característiques en particular. En aquest punt és descriu l'algorisme utilitzat per fer-ho.

Les texture amb OpenGL funcionen amb el punt $(0, 0)$ a la cantonada inferior esquerra i el punt $(1, 1)$ a la cantonada superior dreta. Com que les imatges als sistemes d'Apple funcionen amb el punt $(0, 0)$ a la part superior esquerra i el punt $(1, 1)$ a la inferior dreta, cal fer un mirall en vertical de la imatge abans que l'OpenGL la pugui utilitzar.

Les imatges també acostumen a tenir les dades comprimides i per tant s'han de transformar les dades imprimint-les en el format esperat per l'OpenGL. El següent codi en mostra el procés.

```
- (void) applyTextureImage:(UIImage*)tImage;
{
    // Load images and flip vertically to use as OpenGL texture
    GLuint width = CGImageGetWidth(tImage.CGImage);
    GLuint height = CGImageGetHeight(tImage.CGImage);
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    void *imageData = malloc( height * width * 4 );
    CGContextRef context = CGBitmapContextCreate( imageData, width, height,
                                                  8, 4 * width, colorSpace,
                                                  kCGImageAlphaPremultipliedLast |
                                                  kCGBitmapByteOrder32Big );
    CGColorSpaceRelease( colorSpace );
    CGContextClearRect( context, CGRectMake( 0, 0, width, height ) );

    CGContextTranslateCTM( context, 0, height );
    CGContextScaleCTM( context, 1., -1. );
    CGContextDrawImage( context, CGRectMake(0, 0, width, height), tImage.CGImage );
```

```
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height,
                 0, GL_RGBA, GL_UNSIGNED_BYTE, imageData);

    CGContextRelease(context);

    free(imageData);
}
```

Es veu en les primeres set instruccions com es crea el context on es dibuixarà la imatge en un format que entengui posteriorment l'OpenGL, esborrant-ne el contingut. Les següents tres instruccions s'encarreguen de canviar els eixos per adaptar-los a les coordenades de l'OpenGL, abans de dibuixar-hi la imatge. Per acabar s'envien les dades de la imatge a l'OpenGL.

4.6.2 Textura i Material

Un característica "no desitjada" a l'utilitzar textures és el fet que es barrejen amb l'ultim material utilitzat i això produeix efecte, molt sovint, no desitjats tal com es mostra en la Figura 4.6.

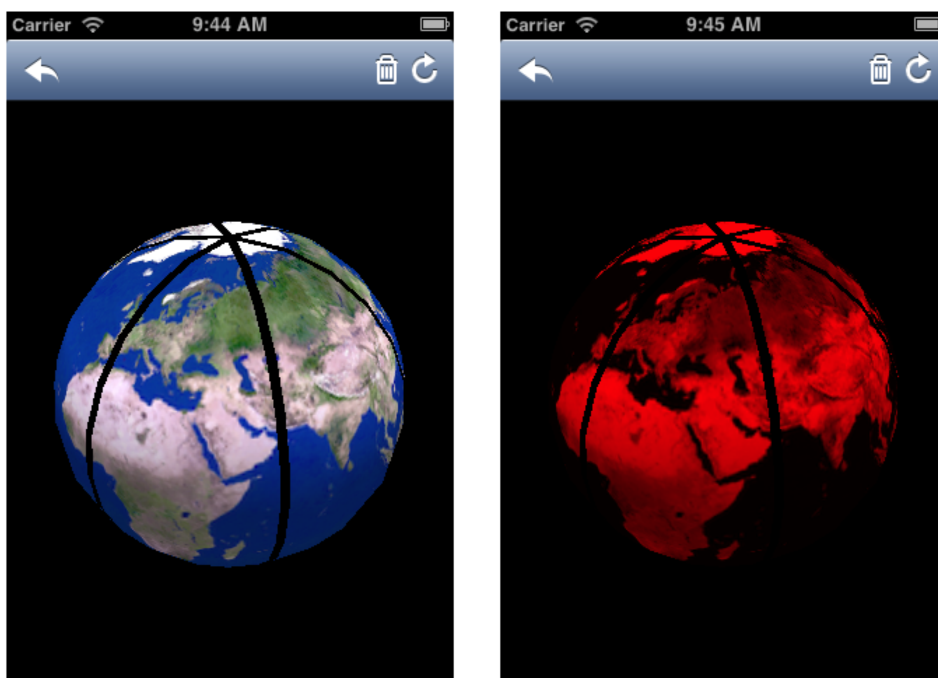

Figura 4.6: Textura sense Material — Textura amb Material Vermell

Per evitar l'efecte de la imatge de la dreta on és veu la terra vermella a causa que s'havia dibuixat anteriorment quelcom de color vermell, se soluciona establint el material amb els materials difús, ambient i especular de color blanc, i el material d'emissió de color negre.

Capítol 5

Anàlisi i Disseny

Aquest capítol se centra en l'anàlisi funcional del simulador, on s'hi identifiquen els diferents tipus d'usuari i els seus casos d'ús agrupats per funcionalitat; i el disseny del simulador tant des del punt de vista del prototipat com el de les classes.

El prototipus està centrat en el 'Mode de Joc' del simulador que és el que s'implementarà. En l'Apèndix B s'hi pot trobar una extensió de l'anàlisi i prototipat amb el 'Mode de Modelat' i el 'Mode de Disseny'. També en l'Apèndix C s'hi pot trobar el disseny per la validació amb usuaris del prototipus.

S'ha estructurat el capítol en les següents seccions:

- **Usuaris**: On s'identifiquen els diferents tipus d'usuari.
- **Casos d'Us**: On es descriuen els casos d'ús de cada tipus d'usuari.
- **Classes**: On es mostra el disseny de classes tant de la interfície d'usuari, com del model.
- **Prototipat**: On es mostra el disseny de les pantalles de la interfície gràfica.

5.1 Usuaris

Tenint present només el 'Mode de Joc' s'ha identificat un únic tipus d'usuari anomenat "Jugador".

5.1.1 Jugador

L'usuari 'Jugador' s'encarrega de manipular els "Cubs de Rubik" canviant-lo de posició i movent-ne les peces. Peces que primer s'han de desordenar per a continuació tornar-les a ordenar. El jugador tracta amb dos mòduls d'ús que es presenten a continuació.

- **Mòdul de Vista**, què s'encarrega de manipular el punt de vista des d'on l'usuari veu el "Cub".
- **Mòdul de Moviments**, què s'encarrega de moure les peces del "cub".

5.2 Casos d'Ús

A continuació es presenten amb més detall els mòduls previstos per al simulador, essent els principals: el mòdul de vista i el de moviments. No obstant en l'Apèndix B s'hi troben altres modes relacionats amb el modelat i disseny de "Cubs de Rubik".

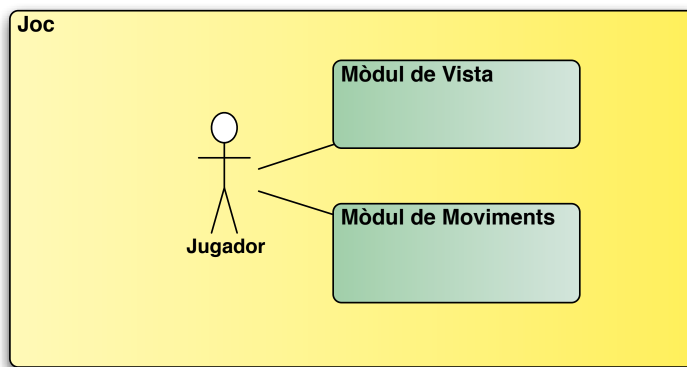

Figura 5.1: Cassos d'Ús — Jugador

Cal afegir, també, que no s'han considerat els cassos de desar i carregar continguts per treballar amb el simulador, ja que es pressuposen. Així mateix la descripció de cada cas d'ús serà superficial ja que s'han analitzat amb més detall en el Capítol 3 sobre el Model de Negoci.

5.2.1 Mòdul de Vista

El mòdul de vista és comú a tots els tipus d'usuaris i ha de permetre visualitzar des de qualsevol angle la figura que se simula. Els seus casos d'ús es mostren en la Figura 5.2.

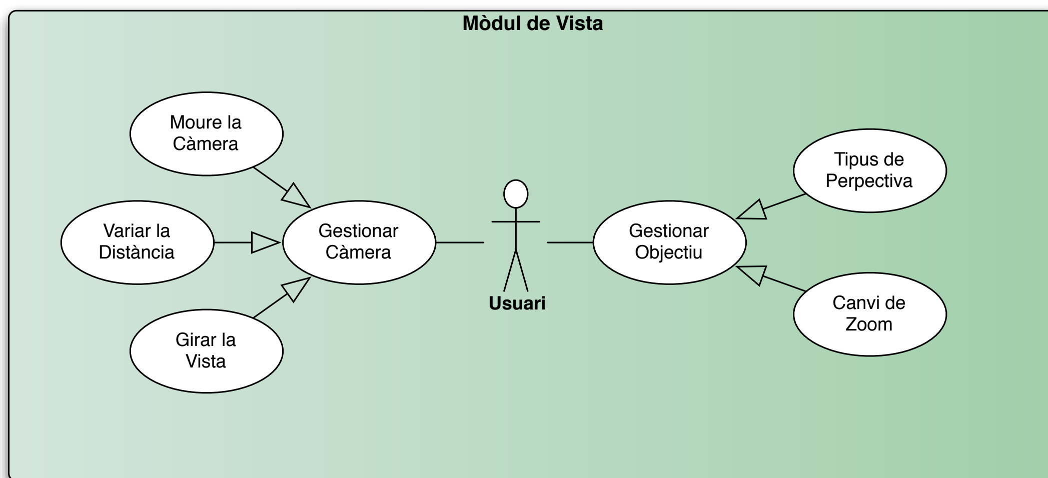

Figura 5.2: Cas d'Ús — Mòdul de Vista

Com que lloc on mira la càmera està situat al centre de la figura, no s'ofereix cap cas d'ús per modificar-lo. Per tant es pot gestionar la càmera movent-la al voltant del centre de la figura, i girant-ne la vista. Per l'altre banda es pot gestionar l'objectiu definint-ne el tipus de perspectiva i canviant-ne el zoom o distància focal.

5.2.2 Mòdul de Moviments

El mòdul de moviments és exclusiu del jugador i li permet moure les peces del "cub" de forma precisa. Els seus casos d'ús es mostren en la Figura 5.3

Entre els casos d'ús s'hi observa els d'identificació de moviment que permet identificar quin nivell es

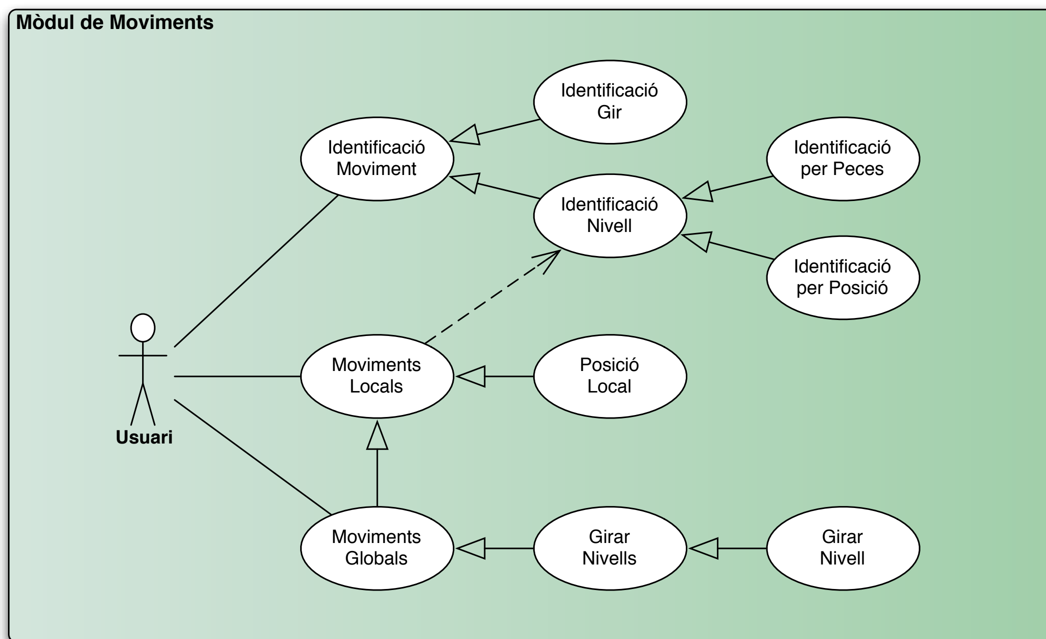

Figura 5.3: Cas d'Ús — Mòdul de Moviments

vol girar i en quin sentit (horari o trigonomètric). La identificació del nivell també es pot fer a partir de les peces que hi pertanyen o per la posició relativa al punt de vista —útil quan s'implementi un sistema de resolució guiat.

Els casos d'ús del moviments locals està pensat per quan s'implementi un sistema guiat de resolució que guiï al jugador en els moviment que ha de fer seleccionant els nivells respecte al seu punt de vista.

Per altra banda hi ha els casos d'us dels moviments global que simplement desplacen les peces dels nivells indicats, se n'estén els cassos específics per fer girs individuals. El cas específic pels moviments locals simplement identifica els nivell per la seva posició i utilitza els moviments globals per a realitzar-lo. També accedeix al cas d'ús de la posició local per poder obtenir els plans de referència i identificar-ne els plans locals dins el sistema global.

5.3 Classes

La programació sobre plataformes Apple, acostuma a seguir el patró de disseny "Model-Vista-Controlador" (MVC), i en aquest projecte també se segueix. Bàsicament el MVC es basa en tenir tots els objectes en una partició basada en tres sub-sistemes, els quals es presenten a continuació.

5.3.1 Vista–Controlador

Ja que les classes de Vista i Controlador són poques es presenten conjuntament per veure com es relacionen entre elles. S'hi pot veure classes tres color diferenciats per a cada sub-sistema: el blau pels controladors, el verd per les vistes i el vermell pel model. També es diferencia amb la intensitat del color si es tracta de classes noves (de color més intens) o classes del sistema (de color pastel).

Com s'ha comentat els controladors s'encarreguen de coordinar la comunicació entre la vista i el model de dades, per a que la vista pugui representar les dades i les dades puguin canviar amb les accions sobre la vista per part de l'usuari. Per aquest motiu els controladors (de color blau) tenen accés tant als elements de la vista com al model.

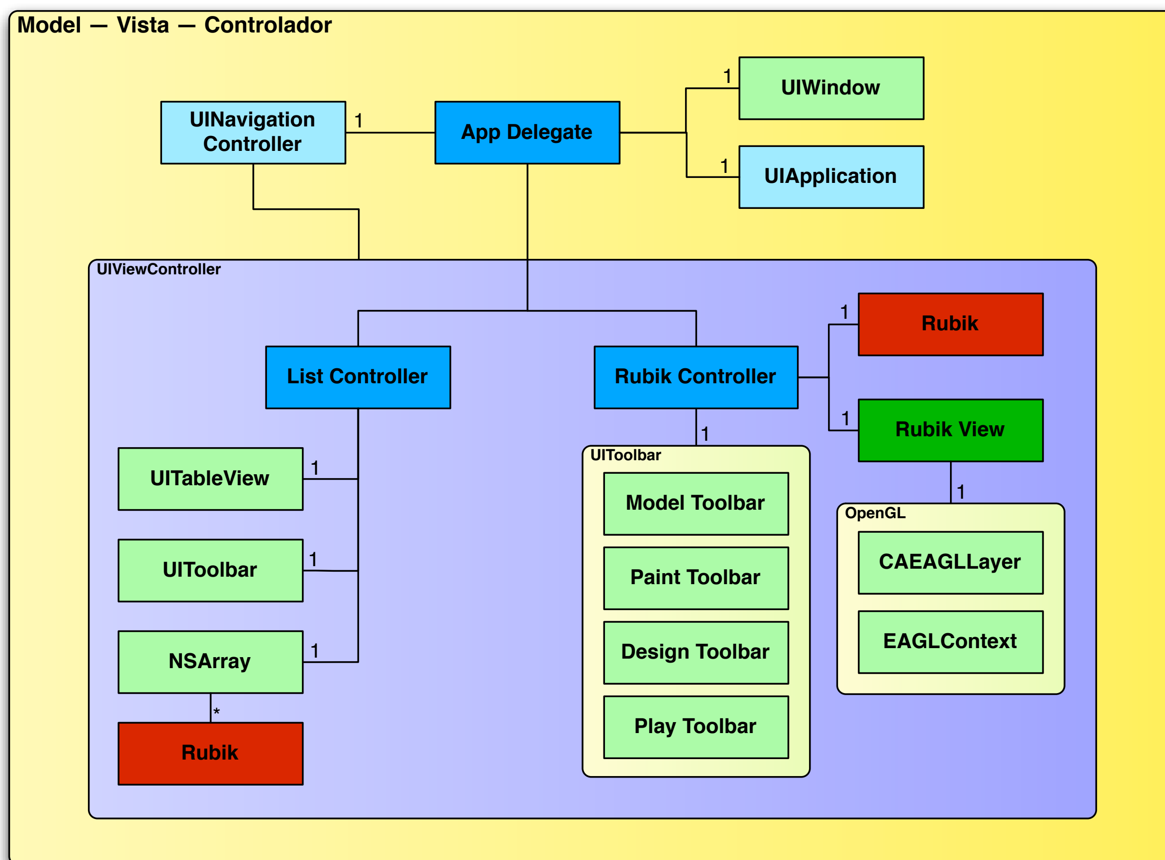

Figura 5.4: Classes — Model-Vista-Controlador

En el diagrama s'hi observen les classes necessàries per a qualsevol aplicació per a iOS: la pròpia aplicació (`UIApplication`) i la finestra on s'hi treballarà (`UIWindow`). D'aquesta en penja l'`AppDelegate` que s'encarrega de gestionar les característiques particulars de l'aplicació, com són, entre altres, la configuració de la navegació entre els controladors: `ListController` i `RubikController`.

La classe `ListController` és el controlador de la pantalla principal que s'encarrega de mostrar el llistat amb tots els "Cubs", per tant conté un array amb els "Cubs", una barra d'eines per poder editar el llistat i afegir nous "Cubs", i per descomptat la taula on es mostren els diferents "Cubs" disponibles.

Quan es prem sobre un "Cub" del llistat es canvia al controlador `RubikController` que s'encarrega de controlar la simulació del "Cub" en tots els seus modes de funcionament: Modelat, Pintat, Disseny i Joc. Per això disposa d'una instància del "Cub de Rubik" (`Rubik`) i la vista `RubikView` que gestiona la presentació en 3D del "Cub" així com la manipulació gestual de l'usuari. També controla cadascuna de les barres d'eines per a cadascun dels modes de funcionament.

5.3.2 Model

El Model, és el sub-sistema que conté els objectes que proporcionen les capacitats i l'emmagatzematge de la informació. Conté totes les regles de processat de les dades. És molt important que aquest sub-sistema sigui plenament funcional sense dependències amb els altres dos.

En el diagrama general que s'ha mostrat en el punt anterior l'única classe que es mostrava del model és la que defineix el "Cub de (`Rubik`). No obstant, tal com mostra la Figura 5.5, aquesta classe no és tant simple, sinó que a dins seu s'hi amaga la potència i complexitat que s'ha mostrat en el Capítol 3 del Model de Negoci.

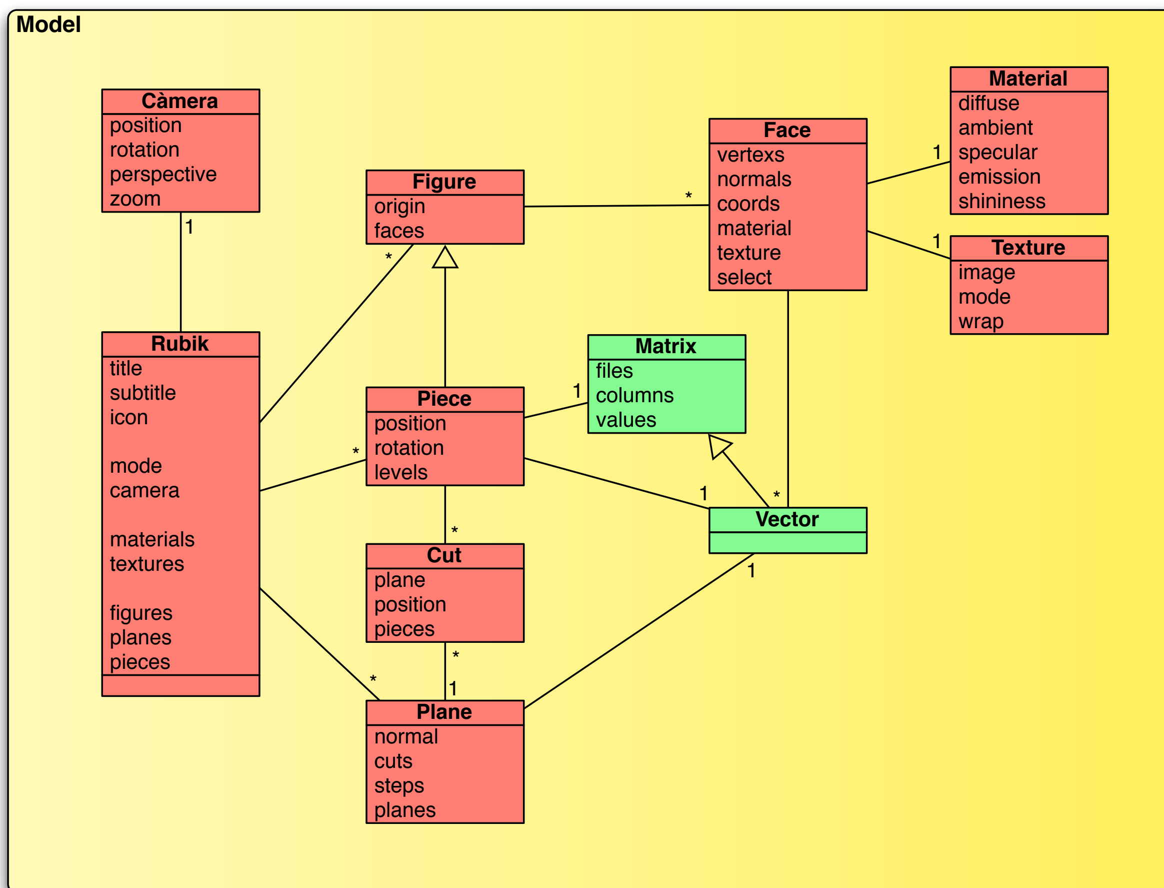

Figura 5.5: Classes Model — El "Cub de Rubik"

En aquest diagrama general del "Cub de Rubik" s'hi observa que la classe `Rubik` conté els elements necessaris per treballar en els diferents modes. Hi ha la `camera` per a mantenir la vista de l'usuari, les `figures` que són una matriu de figures que per a modelar i pintar. També hi ha els `planes` que conté el disseny de talls del "Cub", que en defineix el funcionament; i per acabar les `pieces` que és un array amb totes les peces que es poden moure referenciades amb els plans i talls, i amb la figura que representa la peça.

Les variables `materials` i `textures` conté una referència a tots els materials i textures que utilitza el "Cub". Per últim s'ha afegit una variable `mode` per establir si el "Cub" està en el Mode de Modelat, Disseny o Joc —tot i que només hi ha s'implementarà el Mode de Joc. Així al crear un "Cub" de nou, l'estat passarà a Modelat, quan es crea el primer pla passa a l'estat de disseny i quan es generin les peces del "Cub" passarà a l'estat de joc.

La Figura

S'hi observa la classe `Figure` es veu que, a banda del seu origen, només hi té cares. Cada `Face` està definida pels seus vèrtexs, les seves normals i el seu material —que s'ha previst comú per tots els vèrtexs—, també s'ha previst utilitzar una textura ajustada per les coordenades en cada vèrtex.

La classe `Material` queda definida pels quatre colors: difús, ambient, especular i emissió; juntament amb la brillantor, les quals donen realisme a la figura. La classe `Textura` conté la imatge que la defineix, i opcionalment les dades de cop s'ha d'aplicar sobre la cara.

Els Plans

Els plans basats amb la classe `Plane` estan definits per la normal del pla i els talls —paral·lels a aquest— que s'aplicaran al "Cub" i que n'establiran el seu funcionament. Cada tall representat amb la classe `Cut` està definit pel pla al qual pertany i la distància respecte a aquest on està situat. Quan es relaciona amb les peces d'utilitza la variable `pieces` que conté el llistat de peces que hi ha immediatament a sobre seu.

La classe `Plane` també conté en la variable `planes` la relació sobre el moviment de la resta de plans respecte a ell, dades útils a l'hora del disseny per a comprovar si aquest és correcte.

Matemàtica

De color verd s'identifiquen les classes `Matrix` i `Vector` que s'utilitzen assíduament per a representar punts en l'espai així com a matrius de rotació.

La classe `Vector` hereta directament de la classe `Matrix` ja que és simplement una matriu d'una sola dimensió, que implementa mètodes especialitzats i poca cosa més. Aquestes classes s'utilitzen com a variables en moltes altres classes, i també dins de les classes Vista i Controlador ja que permeten operar sobre l'espai 3D.

Per exemple, un pla està definit per un `Vector` normal. Una cara està definida pels vèrtexs i les normals que són `Vectors`, i fins i tot la posició de la càmera és un `Vector`. En el Capítol 4 s'hi pot trobar una descripció de les tècniques de càlcul geomètric que utilitzen aquestes classes.

5.4 Prototipat

En aquesta secció es presenta l'estructura de pantalles que disposaria una aplicació tipus que pogués interactuar amb el simulador en el Mode de Joc. No obstant en l'Apèndix B s'hi pot trobar una extensió del prototipat per a modes futurs.

5.4.1 Pantalla Principal

La primera pantalla ha de poder treballar amb els diferents "Cubs" que hi hagi instal·lats en l'aplicació. Per tant s'ha pensat en un llistat que mostri els "Cub" que hi ha.

S'ha procurat seguir la guia d'estil d'Apple mantenint en la mesura del possible el disseny comú de les aplicacions sobre iOS. Cada "Cub" està representat per una icona, un nom i una breu descripció. S'ha escollit una alçada de cel·la gran per a que la imatge del "Cub" sigui grossa i pugui identificar-se fàcilment.

Des de la mateixa pantalla principal, en la barra d'eines superior s'hi pot observar un botó per entrar en el mode d'edició. Això permet editar, ordenar i esborrar "Cubs" existents. També hi apareix un botó per crear nous "Cubs".

5.4.2 Plantalla de Joc

Des de la pantalla principal sense editar es pot accedir a la pantalla de joc simplement seleccionant el "Cub" amb que es vulgui jugar.

Aquesta pantalla disposa principalment d'una vista amb el "Cub" per a poder-lo manipular amb els dits. Aquesta pantalla està pensada per manipular-se en posició apaïsat i s'ha situat una barra d'eines a la banda esquerra que permeti barrejar el "Cub", tornar a la posició inicial, i sortir de la simulació per tornar a la pantalla principal.

Figura 5.6: Pantalla inicial: Llistat — Edició — Esborrat

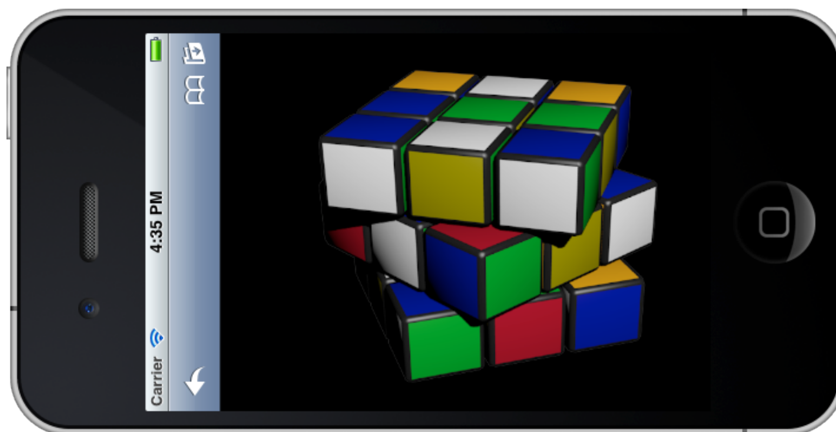

Figura 5.7: Pantalla de Joc

Capítol 6

Implementació

Aquest capítol mostrarà com s'ha implementat el simulador diferenciant-ne el model, la vista i el controlador, així com l'aplicació mínima que el fa servir.

Totes les classes del projecte utilitzen el prefix `RBK9`. Això és un convenció establerta per part d'Apple per mantenir una coherència amb la resta de llibreries del sistema que es diferencien principalment per aquest prefix.

6.1 L'aplicació

La implementació de l'aplicació que utilitza el simulador és realment simple, consta de dues classes que s'encarreguen de gestionar l'aplicació i els "Cubs de Rubik".

6.1.1 RBK9AppDelegate

La classe `RBK9AppDelegate` (delegada de l'aplicació) és una classe heretada de `UIResponder` que implementa el protocol `UIApplicationDelegate` que s'encarrega de gestionar la comunicació que té l'aplicació —que no s'acostuma a utilitzar— amb el sistema operatiu.

Alguns dels missatges que gestiona la delegada de l'aplicació són: els canvis d'estat de l'aplicació, com la senyal que s'ha engegat, que entra en background, que torna a estar activa, que ha de finalitzar, etc; la gestió de l'orientació de l'aparell; la gestió de notificacions remotes i del sistema; etc.

En la nostra aplicació, aquesta classe ha implementat únicament el següent mètode que es crida un cop l'aplicació ha engegat. En aquest mètode es carreguen els "Cubs de Rubik" prèviament desats en un fitxer, deixant-se disponibles a l'aplicació; i es configuren les pantalles inicials de l'aplicació.

```
- (BOOL) application:(UIApplication*)tApplication
            didFinishLaunchingWithOptions:(NSDictionary*)tOptions;
```

6.1.2 RBK9ListController

La classe `RBK9ListController` hereta de `UIViewController` i s'encarrega de controlar la vista que mostrarà el llistat amb els cubs carregats. En els següents punts se'n descriu els mètodes implementats agrupats per funcionalitat.

La interfície gràfica que controla aquest controlador està format per una taula on a cada cel·la s'hi

Figura 6.1: RBK9ListController: Llistat — Edició — Eliminar

mostrarà un cub, una barra d'eines amb els seus botons que permeti en un futur gestionar aquests cubs: afegint-ne, copiant-ne o eliminat-ne.

Accions

La majoria d'elements de la vista tenen associades certes accions. En el nostre cas, els botons quan es premen envien un missatge al controlador que aquest rep i actua en conseqüència.

Així, hi ha els següents mètodes que es criden al prémer el botó d'editar, d'afegit o d'acceptar. Com que l'aplicació no permet crear nous cubs, actualment simplement controlen l'aparença de la barra d'eines per permetre

```
- (IBAction) actionEdit:(id)tSender;
- (IBAction) actionAdd:(id)tSender;
- (IBAction) actionDone:(id)tSender;
```

Orientació

Per controlar les orientacions permeses en aquesta interfície s'utilitzen els següents mètodes que cal sobreescriure de la classes `UIViewController` i que s'encarreguen depenent de la versió del sistema operatiu (pre-iOS6 i iOS6) indicar si un canvi d'orientació en l'aparell a de comportar i gir en la seva interfície. En aquesta aplicació, només es permet la orientació vertical per defecte.

```
- (BOOL) shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)tInterfaceOrientation;
- (BOOL) shouldAutorotate;
- (NSUInteger) supportedInterfaceOrientations;
- (UIInterfaceOrientation) preferredInterfaceOrientationForPresentation;
```

A causa d'un error en el nou sistema operatiu iOS6, s'ha hagut de crear una categoria de la classe `UINavigationController` per a solucionar-lo i permetre que les seves orientacions s'adaptin als controladors que estan mostrant. Aquesta categoria `iOS6Fixes` es troba implementada dins dels fitxers `UINavigationController`iOS6Fixes+.

Origen de Dades de la taula

Per a que la taula mostri els cubs, aquesta delega el seu contingut a un altre objecte que en aquest cas és el propi controlador. Per a que la taula mostri les dades la taula crida els següents dos mètodes del delegat d'origen de dades `UITableViewDataSource` per a obtenir-les.

```
- (NSInteger) tableView:(UITableView*)tTableView
  numberOfRowsInSection:(NSInteger)tSection;
- (UITableViewCell*) tableView:(UITableView*)tTableView
         cellForRowAtIndexPath:(NSIndexPath*)tIndexPath;
```

El primer mètode s'encarrega de retornar el nombre de files que té la taula, i simplement retorna el nombre de "Cubs" que hi ha en el llistat.

El segon dels mètodes retorna la cel·la de la taula indicada per l'index de la fila. Bàsicament la seva implementació s'encarrega de crear una cel·la que mostri una imatge de "cub", un títol i una descripció i el retorna per a que la taula el dibuixi.

Existeixen altres mètodes que permeten configurar-hi seccions, afegir-hi un índex, o preguntar si certa fila pot moure's o editar-se. Mètodes que en aquest cas no cal implementar.

Delegat de la taula

Per a controlar la taula en si, la taula disposa un delegat que ha d'implementar el protocol `UITableViewDelegate` i permet definir l'alçada de les cel·les, controlar-ne les accions, entre d'altres.

En aquest cas, el delegat de la taula segueix sent aquest mateix controlador, i simplement implementa el següent mètode que es crida quan hom prem sobre una cel·la de la taula. Aquest mètode simplement agafa el cub indicat per la posició en la taula, l'afegeix al controlador Rubik que es veu més endavant i canvia de pantalla per poder manipular-lo.

```
- (void) tableView:(UITableView*)tTableView
         didSelectRowAtIndexPath:(NSIndexPath*)tIndexPath;
```

6.2 El Model

Les classes model són les que defineixen tot el model de negoci, i són transparents a la resta de classes, poden funcionar amb altres interfícies gràfiques.

El model conté les classes dels objectes identificats en el model de negoci, així com classes d'ajuda. La classe `RBK9Rubik` és la classe principal del model que defineix un "Cub de Rubik". Al seu interior hi té instàncies de les classes `RBK9Piece`, `RBK9Plane` i `RBK9Cut`; les quals permeten indicar els talls del "Cub" així com les peces que en formen part. La classe `RBK9Camera` permet controlar des d'on es mira el "Cub".

Alhora la classe RBK9Piece utilitza la classe RBK9Figure que defineix la geometria de cada peça utilitzant un conjunt d'objectes de la classe `RBK9Face` que defineix una cara de cert material i/o textura composat per un conjunt de vectors. Totes aquests classes també utilitzen freqüentment les classes `RBK9Matrix` i `RBK9Vector` per poder definir-ne les seves variables.

A continuació se'n descriu cadascuna amb més detall.

6.2.1 RBK9Matrix

La primera classe que s'ha implementat és una classes per treballar amb matrius i poder realitzar-hi les operacions matemàtiques que s'han de fer per efectuar i desar els moviments de les peces del "Cub".

Variables

La definició de la classes que es mostra a continuació bàsicament està formada per el nombre de files i columnes, així com un vector amb els valors de la matriu.

```
@interface RBK9Matrix : NSObject <NSCoding, NSCopying>
{
    NSUInteger  _files;
    NSUInteger  _columns;
    float       *_values;
}
```

Inicialització

Per a crear i/o inicialitzar matrius s'ha definit els següents mètodes, que permeten crear una gran varietat de tipus de matrius. Des de matrius amb tots els seus valors a zero, a un valor establert, o amb tots els valors definits. També es poden crear matrius diagonals.

```
+ (RBK9Matrix*) matrixWithFiles:(NSUInteger)tFiles
                        columns:(NSUInteger)tColumns;
+ (RBK9Matrix*) matrixWithValue:(float)tValue
                          files:(NSUInteger)tFiles
                        columns:(NSUInteger)tColumns;
+ (RBK9Matrix*) matrixWithValues:(float*)tValues
                           files:(NSUInteger)tFiles
                         columns:(NSUInteger)tColumns;
+ (RBK9Matrix*) matrixWithDiagonalValue:(float)tValue
                                  files:(NSUInteger)tFiles
                                columns:(NSUInteger)tColumns;
+ (RBK9Matrix*) matrixWithDiagonalValues:(float*)tValues
                                   files:(NSUInteger)tFiles
                                 columns:(NSUInteger)tColumns;
- (id) initWithFiles:(NSUInteger)tFiles
             columns:(NSUInteger)tColumns;
- (id) initWithValue:(float)tValue
               files:(NSUInteger)tFiles
             columns:(NSUInteger)tColumns;
- (id) initWithValues:(float*)tValues
                files:(NSUInteger)tFiles
              columns:(NSUInteger)tColumns;
- (id) initWithDiagonalValue:(float)tValue
                       files:(NSUInteger)tFiles
                     columns:(NSUInteger)tColumns;
- (id) initWithDiagonalValues:(float*)tValues
                        files:(NSUInteger)tFiles
                      columns:(NSUInteger)tColumns;
```

Mètodes amb un nom prou descriptiu que no cal comentar.

Accés

Per accedir i modificar valors individuals, així com saber la dimensió de la matriu s'han afegit els següents mètodes d'accés a banda dels directes sobre les variables (files, columns, values).

```
- (NSUInteger) size;
- (float) valueAtFile:(NSUInteger)tFile andColumn:(NSUInteger)tColumn;
- (void) setValue:(float)tValue atFile:(NSUInteger)tFile andColumn:(NSUInteger)tColumn;
```

El primer retorna el nombre d'elements de la matriu multiplicant-ne el nombre de files pel nombre de columnes. Els altres dos permeten accedir als valors per fila i columna tant per modificar-ne el valor com per obtenir-lo. En aquestes dues últimes si la fila o columna no existeixen retorna una excepció de tipus `NSRangeException`.

Comparació

Tot i que no és estrictament necessari s'han implementat els mètodes de comparació establint el `hash` amb una XOR els primers 10 valors, i comparant-ne la igualtat per les dimensions i cadascun dels valors amb un cert marge.

```
- (NSUInteger) hash;
- (BOOL) isEqual:(RBK9Matrix*)tObject;
```

Codificació i Còpia

Per poder desar, llegir i copiar una matriu s'han implementat els protocols de codificació `NSCoding` i de còpia `NSCopying` definits amb els següents mètodes.

```
- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
- (id) copyWithZone:(NSZone*)tZone;
```

Operació

Les matrius no només són interessants per l'emmagatzematge ordenat, sinó per les operacions matemàtiques que hi ha associades. Aquesta classe matriu té implementades les operacions més comuns que s'utilitzaran en l'aplicació: suma i reta de valors i matrius, producte de valors i matrius, canvi de signe i matriu trasposta. Cadascuna d'aquestes operacions disposa d'una versió que modifica la pròpia matriu i una versió que en retorna una nova matriu.

```
- (void) minusValue:(float)tValue;
- (void) plusValue:(float)tValue;
- (void) divideValue:(float)tValue;
- (void) multiplyValue:(float)tValue;
- (RBK9Matrix*) matrixMinusValue:(float)tValue;
- (RBK9Matrix*) matrixPlusValue:(float)tValue;
- (RBK9Matrix*) matrixDivideValue:(float)tValue;
- (RBK9Matrix*) matrixMultiplyValue:(float)tValue;

- (void) minus;
- (void) transpose;
- (RBK9Matrix*) matrixMinus;
```

```
- (RBK9Matrix*) matrixTranspose;

- (void) minusMatrix:(RBK9Matrix*)tMatrix;
- (void) plusMatrix:(RBK9Matrix*)tMatrix;
- (void) multiplyMatrix:(RBK9Matrix*)tMatrix;
- (RBK9Matrix*) matrixMinusMatrix:(RBK9Matrix*)tMatrix;
- (RBK9Matrix*) matrixPlusMatrix:(RBK9Matrix*)tMatrix;
- (RBK9Matrix*) matrixMultiplyMatrix:(RBK9Matrix*)tMatrix;
```

No fa falta descriure-les més en detall.

Normalització

Al treballar amb matrius i operari constantment els seus valors acostumen a no ser mai igual a zero. Els següents mètodes permeten normalitzar els zeros si són dins del límit establer pel paràmetre. El primer mètode normalitza la pròpia matriu, mentre que el segon en normalitza una còpia.

```
- (void) normalizeZerosWithLimit:(float)tLimit;
- (RBK9Matrix*) matrixNormalizeZerosWithLimit:(float)tLimit;
```

Impressió

Per a poder depurar les aplicacions és necessari de vegades imprimir el contingut d'un objecte, en aquest cas d'una matriu, per poder-ne veure els seus valors i entendre què passa. Per tant s'ha implementat el mètode estàndard (`description`) per representar en format de cadena de text un objecte

```
- (NSString*) description;
```

Tests

Com tots les classes del Model, cal que aquestes estiguin lliure d'errors tant com sigui possible. Per això, cada classe del model disposa d'una classe de test —`RBK9MatrixTest` en aquest cas— que intenta comprovar i detectar qualsevol error, implementant els següents mètodes i mètodes d'ajuda.

```
- (void) testInit;
- (void) testCompare;
- (void) testCoding;
- (void) testNormalize;
- (void) testOperatingValue;
- (void) testOperatingTranspose;
- (void) testOperatingMatrix;
```

El primer dels testos (`testInit`) comprova els mètodes per crear i inicialitzar matrius. El segon (`testCompare`) comprova els mètodes de comparació. El tercer (`testCoding`) en comprova la codificació. El quart (`testNormalize`) els mètodes de normalització. El cinquè, sisè i setè (`testOperation...`) comprova els diferents mètodes per operar amb matrius.

Per a ajudar en els testos i no haver de fer testos sempre amb matrius dels mateixos valors, s'ha creat els següents mètodes de classes que permet crear matrius amb valors aleatoris amb diferents rangs de valors, files i columnes.

```
+ (RBK9Matrix*) randomMatrix;
```

```
+ (RBK9Matrix*) randomMatrixWithFilesRange:(NSRange)tFiles
                           andColumnsRange:(NSRange)tColumns;
+ (RBK9Matrix*) randomMatrixWithValueRange:(NSRange)tRange
                                filesRange:(NSRange)tFiles
                           andColumnsRange:(NSRange)tColumns;
+ (RBK9Matrix*) randomMatrixWithValueRange:(NSRange)tRange
                                     files:(NSUInteger)tFiles
                                andColumns:(NSUInteger)tColumns;
```

6.2.2 RBK9Vector

La classe `RBK9Vector` és una classe especialitzada de `RBK9Matrix` que treballa amb vector, o el que és el mateix, matrius d'una sola fila o d'una sola columna.

A aquesta classe se li han afegit mètodes de creació i accés adaptats a la seva especificitat, així com operacions només disponibles per a vectors. Mètodes descrits a continuació

Inicialització

Com que els vectors només tenen una dimensió s'han afegit els següents mètodes adaptats a crear-los, poden indicar-ne si són verticals o horitzontals.

```
+ (RBK9Vector*) vectorWithValues:(float*)tValues
                           count:(NSUInteger)tCount
                        vertical:(BOOL)tVertical;
- (id) initWithValues:(float*)tValues
                count:(NSUInteger)tCount
             vertical:(BOOL)tVertical;
```

Accés

Els vectors poden accedir als seus valors simplement indicant-ne l'index, sense tenir en compte si són verticals o horitzontals.

```
- (BOOL) isVertical;
- (float) valueAtIndex:(NSUInteger)tIndex;
- (void) setValue:(float)tValue atIndex:(NSUInteger)tIndex;
```

Operacions

Els vectors se'n pot calcular la distància, i transformar-los a vectors de distància unitat. Aquestes són els mètodes per aconseguir-ho.

```
- (float) distance;
- (void) unit;
- (RBK9Vector*) vectorUnit;
```

Més Operacions

Existeixen més operacions específiques per a vectors, com ho són el producte escalar que pot fer-se amb un vector de qualsevol mida; i el producte vectorial que només es pot fer amb dos vectors de 3

dimensions. En aquest últim cas, si un dels vectors no és de 3 dimensions es llença una excepció de tipus RBK9No3DVectorException.

```
- (float) dotProductWithVector:(RBK9Vector*)tVector;
- (RBK9Vector*) crossProductWithVector:(RBK9Vector*)tVector;
```

Tests

D'igual forma que amb la classe RBK9Matrix, aquesta també disposa d'una classe de text RBK9VectorTest que en comprova la nova funcionalitat amb els següents tests que evidentment comproven els nous mètodes d'inicialització, de comparació, codificació i les noves operacions.

```
- (void) testInit;
- (void) testCompare;
- (void) testCoding;
- (void) testCalculate;
```

Per ajudar en aquests testos, s'han creat mètodes per crear vectors de forma aleatòria.

```
+ (RBK9Vector*) randomVector;
+ (RBK9Vector*) randomVectorWithSizeRange:(NSRange)tSize
                                 vertical:(BOOL)tVertical;
+ (RBK9Vector*) randomVectorWithValueRange:(NSRange)tRange
                              andSizeRange:(NSRange)tSize
                                  vertical:(BOOL)tVertical;
+ (RBK9Vector*) randomVectorWithValueRange:(NSRange)tRange
                                   andSize:(NSUInteger)tSize
                                  vertical:(BOOL)tVertical;
```

6.2.3 RBK9Figure

Per a poder representar gràficament els "Cubs de Rubik" cal que cadascuna de les seves peces tinguin una geometria. Per això hi ha la classe RBK9Figure que bàsicament és un conjunt de cares dibuixades al voltant d'un origen.

```
@interface RBK9Figure : NSObject <NSCoding, NSCopying>
{
    RBK9Vector *_origin;
    NSMutableArray *_faces;
}
```

El funcionament d'aquesta classe és bàsicament de contenidor, i bàsicament s'han implementat els mètodes de creació i inicialització, de codificació i de còpia. Mètodes que han passat els testos de la classe RBK9FigureTest.

```
+ (RBK9Figure*) figureWithFaces:(NSArray*)tFaces;
+ (RBK9Figure*) figureWithFaces:(NSArray*)tFaces andOrigin:(RBK9Vector*)tOrigin;

- (id) initWithFaces:(NSArray*)tFaces;
- (id) initWithFaces:(NSArray*)tFaces andOrigin:(RBK9Vector*)tOrigin;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
- (id) copyWithZone:(NSZone*)tZone;
```

A banda dels mètodes bàsics s'han creat alguns mètodes per a crear poliedres regular que podran ser útils quan es dissenyi el mode de modelat de figures.

```
+ (RBK9Figure*) figureTetrahedronWithScale:(float)tScale;
+ (RBK9Figure*) figureHexahedronWithScale:(float)tScale;
```

Els següents mètodes són útils per a poder trobar els límits i el centre geomètric de la figura, que pot ser útil per filtrar si intersecciona amb d'altres figures, o saber-ne la posició relativa respecte el seu centre.

```
- (NSArray*) cubeCoordinates;
- (RBK9Vector*) centerCoordinate;
```

6.2.4 RBK9Face

Cada cara de la figura està definida mitjançant la classe `RBK9Face` que està definida per un llistat de vèrtexs, un llistat de normals i un possible llistat de coordenades de textura; juntament amb la textura i el material com s'haurà de pintar la cara.

```
@interface RBK9Face : NSObject <NSCoding, NSCopying>
{
    NSMutableArray  *_vertexs;
    NSMutableArray  *_normals;
    NSMutableArray  *_coords;

    RBK9Texture     *_texture;
    RBK9Material    *_material;

    BOOL            _select;
}
```

La variable `_select` és una ajuda que permet al simulador saber si ha de considerar una cara com a seleccionable o no. Així, es procurarà que només siguin seleccionables les cares externes de les peces per evitar errors quan se seleccionen cares internes de les peces que tenen vèrtexs molt pròxims al centre.

En aquesta classe s'han implementat mètodes de creació i inicialització variats, mètodes de comparació, de codificació i de còpia; els quals han estat testejats amb la classe `RBK9FaceTest`.

```
+ (RBK9Face*) faceWithVertexs:(NSArray*)tVertexs;
+ (RBK9Face*) faceWithVertexs:(NSArray*)tVertexs
                      normals:(NSArray*)tNormals
                  andMaterial:(RBK9Material*)tMaterial;
+ (RBK9Face*) faceWithVertexs:(NSArray*)tVertexs
                      normals:(NSArray*)tNormals
                       coords:(NSArray*)tCoords
                   andTexture:(RBK9Texture*)tTexture;
+ (RBK9Face*) faceWithVertexs:(NSArray*)tVertexs
                      normals:(NSArray*)tNormals
                       coords:(NSArray*)tCoords
                      texture:(RBK9Texture*)tTexture
                  andMaterial:(RBK9Material*)tMaterial;

- (id) initWithVertexs:(NSArray*)tVertexs;
- (id) initWithVertexs:(NSArray*)tVertexs
               normals:(NSArray*)tNormals
```

```
           andMaterial:(RBK9Material*)tMaterial;
- (id) initWithVertexs:(NSArray*)tVertexs
               normals:(NSArray*)tNormals
                coords:(NSArray*)tCoords
            andTexture:(RBK9Texture*)tTexture;
- (id) initWithVertexs:(NSArray*)tVertexs
               normals:(NSArray*)tNormals
                coords:(NSArray*)tCoords
               texture:(RBK9Texture*)tTexture
           andMaterial:(RBK9Material*)tMaterial;

- (RBK9Vector*) normal;

- (NSUInteger) hash;
- (BOOL) isEqual:(RBK9Face*)tObject;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
- (id) copyWithZone:(NSZone*)tZone;
```

El mètode `normal` calcula la normal del pla format pels tres primers vèrtexs de la cara, i s'utilitza quan s'inicialitza una cara sense normals definides i que per defecte queden establertes amb aquesta normal calculada.

6.2.5 RBK9Material

Per a poder dibuixar una cara de cert color s'ha definit la classe `RBK9Material` que té com a variables els colors (difús, ambient, emissió i especular) així com la brillantor de l'especular. Aquests colors per a ser vàlids cal que estiguin definits com a RGBA i si no ho estan la classe llença una excepció del tipus `RBK9NoRGBAException`

```
@interface RBK9Material : NSObject <NSCoding>
{
    UIColor *_diffuse;
    UIColor *_ambient;
    UIColor *_emission;
    UIColor *_specular;
    float   _shininess;
}
```

En aquesta classe com en les anteriors s'han definit mètodes d'inicialització, d'accés, de comparació i de codificació que s'han testejat amb la classe `RBK9MaterialTest` per comprar-ne cadascun dels aspectes indicats.

El següent és el llistat de mètodes implementats

```
+ (RBK9Material*) material;
+ (RBK9Material*) materialWithDiffuseAmbientSpecularColor:(UIColor*)tColor
                                             andShininess:(float)tShininess;
+ (RBK9Material*) materialWithDiffuse:(UIColor*)tDiffuse
                              ambient:(UIColor*)tAmbient
                             emission:(UIColor*)tEmission
                             specular:(UIColor*)tSpecular
                         andShininess:(float)tShininess;

- (id) init;
```

```
- (id) initWithDiffuseAmbientSpecularColor:(UIColor*)tColor
                              andShininess:(float)tShininess;
- (id) initWithDiffuse:(UIColor*)tDiffuse
               ambient:(UIColor*)tAmbient
              emission:(UIColor*)tEmission
              specular:(UIColor*)tSpecular
          andShininess:(float)tShininess;

- (void) setDiffuse:(UIColor*)tColor;
- (void) setAmbient:(UIColor*)tColor;
- (void) setSpecular:(UIColor*)tColor;
- (void) setEmission:(UIColor*)tColor;

- (NSUInteger) hash;
- (BOOL) isEqual:(RBK9Material*)tObject;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

La re-definició dels mètodes d'accés com el `setDiffuse:`, s'ha fet per comprovar que el color està definit com a RBGA, i poder llençar l'excepció en cas contrari.

6.2.6 RBK9Texture

Igual que el material, una cara pot tenir definida una textura que oferirà un aspecte més realista i complex, que no pas un simple color o material. Per a poder definir textures s'ha creat la classe `RBK9Texture` que defineix la textura amb una imatge i la forma en que es representarà (repetitiva, estirada, proporcional, ...)

```
@interface RBK9Texture : NSObject <NSCoding>
{
    UIImage *_image;
    int _mode;
    int _wrap;
}
```

Com en la classe `RBK9Material` s'han creat mètodes d'inicialització, de comparació i de codificació; què s'han testejat amb la classe `RBK9TextureTest`. Aquests mètodes queden llistats a continuació.

```
+ (RBK9Texture*) textureWithImage:(UIImage*)tImage;
+ (RBK9Texture*) textureWithImage:(UIImage*)tImage mode:(int)tMode andWrap:(int)tWrap;

- (id) initWithImage:(UIImage*)tImage;
- (id) initWithImage:(UIImage*)tImage mode:(int)tMode andWrap:(int)tWrap;

- (NSUInteger) hash;
- (BOOL) isEqual:(RBK9Texture*)tObject;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

6.2.7 RBK9Piece

La classe `RBK9Piece` és una especialització de la classe `RBK9Figure` —tot i que potser seria millor que en comptes d'heretar-la en contingués una instància— que s'amplia amb les variables que indiquen la

posició i orientació de la peça dins el “Cub” i als nivells a que pertany en cada moment.

```
@interface RBK9Piece : RBK9Figure <NSCoding>
{
    RBK9Vector *_position;
    RBK9Matrix *_rotation;
    NSMutableDictionary *_levels;
}
```

La variable `_position` és un vector amb la posició actual del centre de la peça, vector que s'actualitza en cada moviment i que serveix per a calcular-ne la posició ens els nivells del “Cub”. Al començar aquest vector s'inicialitzar amb la coordenada central de la peça, situada en la seva posició per defecte.

La variable `_rotation` és una matriu quadrada de 3x3 que conté la rotació actual de la peça. Variable que serveix per dibuixar la peça en el lloc i orientació correcte. Al començar aquesta matriu és una matriu identitat que indica que no s'ha fet cap rotació.

Per últim la variabel `_levels` és un diccionari que té com a claus tots els plans definits en el “Cub” i com a objectes desats el tall immediatament inferior del nivell on és la peça. Això permet saber amb rapidesa a quins nivells pertany la peça.

Com en la resta de classes s'ha definit una sèrie de mètodes bàsics per a poder inicialitzar i codificar la peça; els quals s'ha testejat amb la classe de test `RBK9PieceTest`.

```
+ (RBK9Piece*) pieceWithFigure:(RBK9Figure*)tFigure;
+ (RBK9Piece*) pieceWithFigure:(RBK9Figure*)tFigure andPlanes:(NSArray*)tPlanes;

- (id) initWithFigure:(RBK9Figure*)tFigure;
- (id) initWithFigure:(RBK9Figure*)tFigure andPlanes:(NSArray*)tPlanes;

- (void) setPlanes:(NSArray*)tPlanes;
- (void) setPosition:(RBK9Vector*)tPosition;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

Cal esmentar la re-definició del mètode `setPlanes:` que aprofita aquest canvi per canviar els valors de la variable `_levels` amb els nous plans. La re-definició del mètode `setPosition:` permet definit el vector de posició com un vector horitzontal si no és el cas per a poder-hi operar posteriorment sense més comprovacions.

A més s'ha implementat el següent mètode que calcula la posició de la peça en cadascun dels nivells de cada pla, així com actualitzar les peces que hi ha a cada pla. Aquest mètode es crida després de moure una peça i actualitza l'estat de peces i nivells respectivament. Aquest mètode també s'ha testejat.

```
- (void) calculateLevels;
```

6.2.8 RBK9Plane

Ja s'ha vist en el capítol sobre el Model de Negoci, què era un pla dins dels “Cubs de Rubik”. I vet aquí la seva definició en la classe `RBK9Plane`

```
@interface RBK9Plane : NSObject <NSCoding,NSCopying>
{
    RBK9Vector *_normal;
```

```
    NSMutableArray *_cuts;

    NSUInteger _steps;
    NSMutableArray *_planes;
}
```

Un pla —tal com s'ha comentat en el capítol de Tècnica i Tecnologia— queda definit per una normal perpendicular a aquest i aquesta és la funció de la variable `_normal`. A la vegada, cada pla té definits en la variable `_cuts` els talls paral·lels a aquest que tindrà el "Cub".

La variable `_steps` indica el nombre de moviments que cal per fer una volta completa respecte aquest pla en qualsevol dels seus nivells. Dividint una volta completa (360°) a aquest valor se sap l'interval permès de gir.

La variable `_planes` —que actualment no s'utilitza— és un conjunt de grups de plans que es generen a cada pas. Aquesta variable ha de facilitar la validació dels plans d'un "Cub" comparant-ne les seves relacions.

Aquesta classe ha implementat i testejat els següents mètodes d'inicialització, accés, comparació, codificació i còpia.

```
+ (RBK9Plane*) planeWithNormal:(RBK9Vector*)tNormal;

- (id) initWithNormal:(RBK9Vector*)tNormal;

- (void) setNormal:(RBK9Vector*)tNormal;
- (void) setCuts:(NSMutableArray*)tCuts;

- (NSUInteger) hash;
- (BOOL) isEqual:(RBK9Plane*)tObject;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
- (id) copyWithZone:(NSZone*)tZone;
```

Cal destacar que la re-definició del mètode `setNormal:` s'ha fet per assegurar-se que la normal és un vector unitat. La re-definició del mètode `setCuts:` permet ordenar els talls de més petit a més gran i afegir-hi un tall en la posició $-\infty$ que servirà per relacionar-lo amb el nivell inferior, ja que cada tall representa el nivell de sobre seu.

Per ajudar a treballar amb els plans i els seus talls, s'han implementat els següents mètodes de càlcul. Mètodes que també s'ha testejat.

```
- (float) angleWithPlane:(RBK9Plane*)tPlane;
- (float) distanceToPoint:(RBK9Vector*)tPoint;
- (RBK9Cut*) cutBelowPoint:(RBK9Vector*)tPoint;
- (RBK9Vector*) intersectionWithPlane:(RBK9Plane*)tPlane;
```

El mètode `angleWithPlane:` permet calcular l'angle —el més petit— entre dos plans qualsevol, i poder decidir si es poden considerar el mateix, o per exemple si dos plans poden generar-se per revolució respecte a un tercer.

El mètode `distanceToPoint:` permet calcular la distància d'un punt en l'espai respecte la superfície del pla. Aquest mètode s'utilitzar per calcular en quin nivell d'aquest pla està situat un punt qualsevol, i per tant saber en quin nivell són les peces del "Cub".

El mètode `cutBelowPoint:` retorna el tall per sota del punt passat com a paràmetre que representa al nivell on és el punt. Aquest mètode en calcula la distància i troba el primer tall que hi ha per sota seu.

L'últim mètode `intersectionWithPlane:` retorna el vector d'intersecció entre dos plans i és útil per a calcular un nou pla perpendicular a aquests dos.

6.2.9 RBK9Cut

Per definir cadascun dels talls i alhora treballar amb els nivells del "Cub" tenim la classe `RBK9Cut` que representa un tall d'un pla (`_plane`) i està definit per aquest pla, i la separació (`_position`) respecte al seu origen $(0, 0, 0)$.

```
@interface RBK9Cut : NSObject
{
    RBK9Plane *_plane;
    float _position;

    NSMutableSet *_pieces;
}
```

A més, en la variable `_pieces` s'hi desa la relació de peces que actualment hi ha immediatament sobre aquest tall —o dit altrament, dins del nivell de sobre el tall— per a poder-hi accedir ràpidament i fer els moviment amb totes les seves peces. Aquesta variable s'actualitza automàticament cada cop que es calculen els nivells de cada peça.

Aquesta classe és un simple contenidor i per tant només s'han implementat i testejat mètodes de creació i inicialització, i de codificació.

```
+ (RBK9Cut*) cutBottomWithPlane:(RBK9Plane*)tPlane;
+ (RBK9Cut*) cutWithPlane:(RBK9Plane*)tPlane andPosition:(float)tPosition;

- (id) initWithPlane:(RBK9Plane*)tPlane andPosition:(float)tPosition;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

6.2.10 RBK9Camera

En el capítol del Model de Negoci s'ha explicat que per mirar el "Cub" en comptes de moure'l el que es faria és canviar el punt de vista des d'on es mira. Per això s'ha creat la càmera `RBK9Camera` definida a continuació.

```
@interface RBK9Camera : NSObject <NSCoding>
{
    RBK9Vector *_position;
    float _rotation;
    float _zoom;
    BOOL _perspective;
}
```

La càmera queda definida per la posició d'aquesta mitjançant un vector en 3D, juntament amb la variable `_rotation` que indica la rotació de la càmera respecte la vertical. Així una rotació de 0° indica que la part superior del "Cub" sempre és veu a sobre, excepte quan es mira zenitalment que a sobre la vista hi ha l'eix Y.

NOTA: Aquesta forma de definir la càmera duu a dificultats de manipulació per part de l'usuari ja que quan es vol mirar el cub per l'altre costat passant per la part zenital, fa un gir brusc per mantenir

la part superior a sobre. Això es pot solucionar creant un nou vector que indiqui la part superior i que al moure la càmera aquest també giri.

Les altres dues variables permeten definir paràmetre focals de la càmera, com són el `_zoom` i el tipus de `_perspective`. Tot i que en principi sempre mostra el cub sencer i no s'acostuma a modificar.

NOTA: Simulant el funcionament del "Cub" la càmera sempre mira en direcció al centre d'aquest. Tanmateix, en "Cubs" més grossos i amb més talls potser caldria poder ampliar el zoom i moure la direcció on es mira per poder centrar-lo en el punt que es vol manipular. També seria interessant aquesta opció quan s'implementi el mode de Modelat per poder treballar amb precisió sobre una part de la figura.

Aquesta classe és una simple classe contenidor i s'han implementat i testejat mètodes d'inicialització i codificació.

```
+ (RBK9Camera*) cameraWithPosition:(RBK9Vector*)tPosition
                   andPerspective:(BOOL)tPerspective;
+ (RBK9Camera*) cameraWithPosition:(RBK9Vector*)tPosition
                         rotation:(float)tRotation
                             zoom:(float)tZoom
                   andPerspective:(BOOL)tPerspective;

- (id) initWithPosition:(RBK9Vector*)tPosition
         andPerspective:(BOOL)tPerspective;
- (id) initWithPosition:(RBK9Vector*)tPosition
               rotation:(float)tRotation
                   zoom:(float)tZoom
         andPerspective:(BOOL)tPerspective;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

6.2.11 RBK9Rubik

Per acabar amb les classes model, hi ha la classe `RBK9Rubik` que representa un "Cub de Rubik" i que està definit a continuació.

```
@interface RBK9Rubik : NSObject <NSCoding>
{
    NSString *_title;
    NSString *_subtitle;
    UIImage *_image;

    RBK9RubikStatus _status;
    RBK9Camera *_camera;

    NSMutableArray *_materials;
    NSMutableArray *_textures;

    NSMutableArray *_figures;
    NSMutableArray *_planes;
    NSMutableArray *_pieces;
}
```

S'hi pot veure un primer conjunt de variables que descriuen el "Cub" amb un títol o nom, un subtítol o descripció, i una imatge representativa. Un segon grup de variables en descriu l'estat o mode de funcionament (Modelat, Disseny i Joc) que per ara sempre està en 'Joc', i la variable `_camera` que indica des d'on es mira el "Cub".

A continuació hi ha definides les variables `_material` i `_textures` que agrupen tots els materials i textures del "Cub" i que han de permetre optimitzar el dibuixat en OpenGL ja que la càrrega de textures té un alt cost computacional.

La variable `_figures` conté el conjunt de figures amb que es treballa en mode de Modelat i que per ara no s'utilitza. Igual que la variable `_planes` que conté els plans i talls amb que es treballa en el mode de Disseny, i que tampoc s'utilitzen.

Per acabar la variable `_pieces` conté la relació de totes les peces juntament amb els plans i talls amb que treballen. I són amb les que es treballa en el mode de Joc.

En aquesta classe s'han implementat i testejat els mètodes bàsics d'inicialització i codificació.

```
+ (RBK9Rubik*) rubikWithTitle:(NSString*)tTitle
                     subTitle:(NSString*)tSubtitle
                     andImage:(UIImage*)tImage;

- (id) initWithTitle:(NSString*)tTitle
            subTitle:(NSString*)tSubtitle
            andImage:(UIImage*)tImage;

- (id) initWithCoder:(NSCoder*)tCoder;
- (void) encodeWithCoder:(NSCoder*)tCoder;
```

A més, s'han implementat el mètode elemental per girar un nivell de peces de cert pla un interval de volta tenint en compte la resposta.

```
- (BOOL) turnLevel:(NSUInteger)tLevel
           ofPlane:(RBK9Plane*)tPlane
        anInterval:(float)tInterval
        withMagnet:(RBK9RubikMagnet)tMagnet;
```

Aquest métode utilitza el número de nivell i el pla per trobar el tall que representa el nivell i en conseqüència les peces que cal moure. A continuació i depenent del tipus de magnetització (`tMagnet`) calcula l'angle final del moviment —així un interval de -0.0001 amb magnetització per sentit calcularà l'angle per girar en sentit negatiu fins a la pròxima posició estable. Per acabar gira cadascuna de les peces l'angle indicat actualitzant-ne la nova posició, rotació i els nivells on ara pertany la peça.

Per poder girar una peça un cert angle respecte a un eix cal calcular una matriu que es multiplicarà a la posició i rotació actual de la peça. Aquesta matriu s'obté amb el següent mètode. El càlcul realitzat per a trobar aquesta matriu està descrita en el Capítol 4.

```
+ (RBK9Matrix*) rotationMatrixWithVector:(RBK9Vector*)tVector
                                   angle:(float)tAngle;
```

Ambdós mètodes anteriors estan testejats adequadament en la classe `RBK9RubikTest`.

Test

La classe de test `RBK9RubikTest` a banda de comprovar el correcte funcionament dels diferents mètodes, té implementats dos mètodes que serveixen per generar manualment certs "Cubs" i desar-los en un fitxer que posteriorment s'afegiran a l'aplicació per poder-los simular.

```
+ (RBK9Rubik*) rubikCube2x2x2;
+ (RBK9Rubik*) rubikEarth3x2;
```

El mètode `rubikCube2x2x2` genera un Cub de Rubik clàssic de 2x2x2 amb les peces cúbiques i les cares amb materials plans. En canvi el mètode `rubikEarth3x2` genera un "Cub de Rubik" esfèric amb 3 talls que el divideixen en 6 peces com si fos una taronja, amb una textura que simula la Terra.

6.3 La Vista

Les classes de vista són les encarregades de mostrar contingut en pantalla, per als casos més generals ja estan implementades amb les vistes de taula, de text, els botons, etc.; però si cal aconseguir un resultat un mica més fora del comú cal implementar-la per un mateix.

En aquest projecte s'ha de dibuixar un "Cub de Rubik" i manipular-lo amb els dits, i evidentment, no hi ha cap vista que faci aquesta funcionalitat. Per tant s'ha creat la classe `RBK9RubikView` que hereta d'un vista base `UIView` per a oferir aquestes característiques al programador.

La definició i funcionalitat d'aquesta classe és descriu amb més detall en els següents punts.

6.3.1 La definició

Com ja s'ha dit, la classe `RBK9RubikView` hereta de la classe del sistema `UIView`, la qual ja disposa d'una funcionalitat bàsica. Per tenir una vista que utilitza OpenGL, a partir de la versió iOS5 ja existeix una classe (`GLKView`) que ja té part de la configuració fet, s'ha preferit fer-la des de zero per poder utilitzar l'aplicació en dispositius més antics.

```
@interface RBK9RubikView : UIView
{
    EAGLContext  *_context;
    GLuint       _frameBuffer;
    GLuint       _depthBuffer;
    GLuint       _colorBuffer;
    GLubyte      *_selectBuffer;

    GLuint       _texture;

    BOOL         _movingCamera;

    id<RBK9RubikViewDelegate> _delegate;
}
```

Mirant la definició de les variables de la classe `RBK9RubikView` s'hi observa amb diferència la variable `_context` que és on l'OpenGL hi dibuixarà.

A continuació s'hi troben tres buffers (`_frameBuffer`, `_depthBuffer` i `_colorBuffer`) que són els que l'OpenGL utilitzarà per pintar l'escena abans d'enviar-la a pantalla. El `_frameBuffer` s'encarrega de contenir-hi els altres dos buffers, el `_depthBuffer` conté la informació de nivell dels objectes pintats i per tant controlar què es dibuixa al davant i què al darrera, i per acabar el `_colorBuffer` conté els colors resultants del dibuixat.

Un altre buffer d'un altre tipus és el `_selectBuffer` que contindrà a cada punt de la pantalla quina peça i cara d'aquesta hi ha dibuixada, així quan se seleccioni un punt de la pantalla immediatament es podrà saber la peça i la cara que s'ha seleccionat.

Per acabar i ha la variable `_texture` que hi conté la textura que s'utilitza en aquell moment (en un futur caldrà modificar aquesta variable per un array de textures per poder controlar més d'una textura alhora).

La variable booleana _movingCamera indica quan l'usuari està movent la càmera. És una variable necessària per a poder detectar el final de certs gestos fets amb diferent quantitats de dits, ja que al finalitzar-se un moviment sempre només hi queda un dit.

Per acabar hi ha la variable _delegate que serà un objecte delegat que adaptarà el protocol RBK9RubikViewDelegate que es comentarà en la Secció 6.3.7.

6.3.2 Activar l'OpenGL

Per activar l'OpenGL en la vista el primer que cal és definir la capa (CALayer) que tindrà la vista mitjançant el mètode de classe següent.

```
+ (Class) layerClass;
{
    // This is mandatory to work with CAEAGLLayer in Cocoa Framework.
    return [CAEAGLLayer class];
}
```

Paral·lelament cal configurar-la la capa com a opaca —per a que sigui més eficient— i crear el context OpenGL os s'hi dibuixarà. Això s'ha de fer en els mètodes initinitWithFrame: per quan es crei la vista per codi, i en el mètode de codificació initWithCoder: per quan es carregui des d'un fitxer tipus XIB.

```
- (void) configureLayer;
{
    [[self layer] setOpaque:YES];
    [self setContext:[[[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES1] autorelease]];
    [EAGLContext setCurrentContext:_context];
}
```

6.3.3 Configurar l'OpenGL

Un cop es té la vista configurada cal configurar l'OpenGL, definit els seus buffer adequadament. Això caldrà fer-ho a quan es crei la vista, o quan aquesta canviï de mida i, per tant, calgui modificar la mida dels buffers. Per això tenim els mètodes següents que creen els buffers i els eliminen, quan sigui el cas.

```
- (void) configureOpenGL;
- (void) deleteBuffers;
```

Com que el buffer _selectBuffer, tot i no ser de l'OpenGL, com que també va relacionat amb la mida de la vista, es crearà i eliminarà dins d'aquests mètodes.

6.3.4 Dibuixar el "Cub"

Amb la vista preparada per dibuixar amb OpenGL i els buffers a punt, la vista oferirà els següents mètodes per dibuixar les peces dels "Cubs de Rubik" per part d'un controlador.

```
- (void) configDraw;
- (void) applyColor:(UIColor*)tColor;
- (void) applyMaterial:(RBK9Material*)tMaterial;
- (void) applyTexture:(RBK9Texture*)tTexture;
- (void) applyTextureImage:(UIImage*)tImage;
- (void) applyCamera:(RBK9Camera*)tCamera;
```

```
- (void) drawClear:(UIColor*)tColor;
- (void) drawPiece:(RBK9Piece*)tPiece rotate:(RBK9Matrix*)tRotate;
- (void) drawFace:(RBK9Face*)tFace;
- (void) applyRender;
```

El mètode `configDraw` s'encarrega de configurar l'OpenGL per a que dibuixi amb llums i textures. Tot i que això podria configurar-se juntament amb els buffers, el fet que a la Secció 6.3.5 per seleccionar peces es desconfiguri, cal tornar-ho a configurar.

Els mètodes `applyColor`, `applyMaterial` i `applyTexture` estableixen el color (sense ús), el material i/o la textura que s'utiltizarà a partir d'aquell moment. Cal dir que l'opció d'aplicar textura té un cost computacional alt i s'acostuma a fer només a l'inici si s'ha desconfigurat (en futures revisions caldrà adaptar-ho a més textures i que pugui modificar-se mentre es dibuixa sense massa increment computacional.

Per començar a dibuixar cal definir l'escena indicant on és el punt de vista i això és fa amb el mètode `applyCamera:` que configura la projecció de l'OpenGL adequadament. Un cop definit el punt de vista cal esborrar l'escena anterior amb el mètode `drawClear:` que esborra el contingut del buffer deixant-ho tot amb el color que se li passa per paràmetre.

A partir d'aquí ja es pot començar a dibuixar peça a peça mitjançant el mètode `drawPiece:rotate:` on el primer paràmetre és la peça a dibuixar i el segon si s'indica és una matriu de rotació que afegeix un moviment extra a la posició actual de la peça. Això permet fer animacions de moviment de les peces sense modificar directament la posició de la peça fins que el moviment hagi finalitzat. Dins d'aquest mètode, s'actualitza la matriu de dibuix amb la matriu de rotació de la peça i el seu origen, per a continuació dibuixar cadascuna de les cares cridant al mètode `drawFace:`.

El mètode `drawFace:` només s'acostuma a cridar des de dins de `drawPiece:rotate:` i bàsicament defineix l'array de coordenades dels vèrtexs, les normals i les textures, així com aplicar-hi el material o la textura adequada a la cara. Un cop configurats els array, en dibuixa els triangles. I per acabar allibera la memòria aquests arrays.

Un cop totes les peces dibuixades, el `_colorBuffer` conté l'escena dibuixada i només cal cridar al mètode `applyRender` per a mostrar-ho sobre la vista.

6.3.5 Seleccionar les Peces

De manera semblant al dibuixat de les peces, per crear el buffer de selecció també s'utilitzar l'OpenGL, el qual en comptes de pintar amb colors i textures hi pinta el nom (un número) de la peça i la cara.

Els mètodes utilitzats per "seleccionar" les peces són els següents, que com es pot observar són molt semblants (alguns iguals) als utilitzats per dibuixar l'escena.

```
- (void) configSelect;
- (void) applyName:(NSUInteger)tName;
- (void) applyCamera:(RBK9Camera*)tCamera;
- (void) selectClear:(NSUInteger)tName;
- (void) selectPiece:(RBK9Piece*)tPiece withName:(NSUInteger)tName;
- (void) selectFace:(RBK9Face*)tFace withName:(NSUInteger)tName;
- (void) applySelect;
```

El mètode `configSelect` com el seu germà `configDraw` s'encarrega de configurar l'OpenGL desactivant-ne els llums i les textures per a que pinti els noms (que seran colors) de forma plana.

A diferència del dibuixat on s'hi aplicaven colors, materials i textures, en aquest cas s'hi aplica un nom amb el mètode `applyName:`. Nom que es codifica com un color que dibuixarà l'OpenGL en el lloc de cada peça.

Com a l'hora de dibuixar l'escena, cal definir el punt de vista amb el mètode `applyCamera:`, i esborrar el buffer ficant-lo tot a zero mitjançant el mètode `selectClear:`

Per pintar la selecció de forma semblan al pintar l'escena es crida per a cada peça el mètode `selectPiece:withName:` que se li passa com a paràmetre la peça a seleccionar i el nom d'aquesta. A dins d'aquest mètode de forma semblant al pintat es modifiquen les matrius de transformació amb la matriu de rotació i l'origen de la peça, i a continuació es passa a seleccionar cadascuna de les cares, combinant-ne el seu número amb la de la peça que ja es té, cridant al mètode `selectFace:withName:`.

Per a cada cara el mètode `selectFace:withName:` crear un array per les coordenades dels vèrtex, hi aplica el nom amb el mètode `applyName:` i passa a "seleccionar" la cara.

Finalment, un cop seleccionades totes les cares, amb el mètode `applySelect` es passen tots els colors al `_selectBuffer` per a poder-hi accedir en qualsevol moment.

Invalidar el buffer

El `_selectBuffer` és un buffer amb els noms de les peces i cares que hi ha en cada punt de la pantalla, i per tant quan la càmera o una peça es mou aquest buffer queda invalidat. Això és fa mitjançant els següents mètodes que s'encarreguen de posar el nom del primer punt de la pantalla amb el nom `0xFFFFFFFF` que no es donarà mai.

```
- (BOOL) selectIsValid;
- (void) invalidateSelect;
- (NSUInteger) selectNameAtPoint:(CGPoint)tPoint;
```

Així no cal imprimir la selecció a cada moviment, sinó només quan algú ho necessita i el buffer és invàlid, estalviant utilització de la CPU.

6.3.6 Configurar els Gestos

Per a poder utilitzar gestos tàctil sobre la vista de Rubik cal definir-los de bon començament. En aquesta vista es controlen els moviments amb dos dits per a canviar el punt de vista tant desplaçant-los (Pan) com girant-los (Rotation), i amb un sol dit desplaçant-los.

Per a capturar els gestos aquests es configuren a l'inicialitzar la vista cridant al mètode `configureGestures`, on es creen, configuren i s'afegeixen per a que els reconegui i cridi als mètodes adequats en cada cas.

```
- (void) configureGestures;
```

6.3.7 Controlar els Gestos

Un cop configurats els gestos, quan se'n produeix qualsevol el sistema crida el següents mètodes que s'encarreguen de gestionar-los.

```
- (void) gesturePan:(UIPanGestureRecognizer*)tGesture;
- (void) gestureRotation:(UIRotationGestureRecognizer*)tGesture;
```

Simplificant, aquests dos mètodes detecten l'inici del gest, el canvi o moviment i el seu final. En el cas de desplaçament (`gesturePan:`), se'n comprova el nombre de dits utilitzats i si són dos dits s'activa la variable `_movingCamera` per tenir constància que s'està realitzant un moviment de càmera i no una selecció.

Un cop recopilada aquesta informació se n'informa al delegat que ha adoptat el protocol `RBK9RubikViewDelegate`, el qual s'encarregarà d'actuar en conseqüència movent el punt de vista o desant la informació dels elements seleccionats per a poder realitzar moviments de peces.

El protocol `RBK9RubikViewDelegate` defineix els següents mètodes

```
- (void) rubikViewWillBeginRotateCamera:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didRotateCameraRadians:(float)tRadians;
- (void) rubikView:(RBK9RubikView*)tView didEndRotateCameraRadians:(float)tRadians;

- (void) rubikViewWillBeginMoveCamera:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didMoveCameraVector:(CGPoint)tVector;
- (void) rubikView:(RBK9RubikView*)tView didEndMoveCameraVector:(CGPoint)tVector;

- (void) rubikViewWillBeginSelect:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didSelectPiece:(NSUInteger)tPiece
                                              andFace:(NSUInteger)tFace;
- (void) rubikView:(RBK9RubikView*)tView didEndSelectPiece:(NSUInteger)tPiece
                                                 andFace:(NSUInteger)tFace;
```

6.4 El Controlador

En el patró de disseny Model-Vista-Controlador (MVC) el controlador s'encarrega de sincronitzar la informació entre el Model i la Vista i viceversa.

En aquest cas, aquest controlador s'encarrega de controlar una vista de Rubik (`RBK9RubikView`) enllaçada amb la variable `_rubikView`, juntament amb una barra d'eines amb els botons del mode de Joc.

NOTA: Quan s'implementin els modes de Modelat i Disseny, aquest controlador controlarà també altres barres d'eines per defecte d'aquests altres modes, intercanviar-les quan es canviï de mode.

Una especialització de la classe podria controlar altres elements dins la vista que en millorin l'experiència d'usuari.

6.4.1 Definició

La definició de la classe mostra que hereta d'una classe `UIViewController` i que adopta el protocol `RBK9RubikViewDelegate` que respon als missatge del la Vista de Rubik (`RBK9RubikView`).

```
@interface RBK9RubikController : UIViewController <RBK9RubikViewDelegate>
{
    // Outlets
    RBK9RubikView  *_rubikView;
    UIToolbar      *_toolBar;

    // Datas
    RBK9Rubik      *_rubik;
    NSMutableArray *_selectPieces;
    NSMutableArray *_selectFaces;

    // Timers
    NSTimer   *_timerRotate;
}
```

Les primeres 2 variables estan enllaçades amb la vista de Rubik, la barra d'eines per defecte per poder canviar-ne els botons depenen del mode de funcionament.

Les següents 3 variables contenen el "Cub" que està controlant, i dos arrays amb les últimes peces diferents que s'han seleccionat juntament amb la cara seleccionada. Aquest dos arrays ens permetrà filtrar quin nivell i de quin pla s'ha de moure.

Per acabar hi ha un temporitzador que s'encarrega de controlar l'animació dels moviments.

6.4.2 Accions de la Interfície

Els botons de les barres d'eines ens ofereixen la possibilitat de manipular el "Cub" i controlar-ne la vista cridant als següents mètodes.

```
- (IBAction) actionExit:(id)tSender;
- (IBAction) actionReset:(id)tSender;
- (IBAction) actionShuffle:(id)tSender;
```

El mètode `actionExit:` s'executa quan es prem el botó de sortir, i simplement torna a mostrar el llistat dels "Cubs' de l'aplicació.

El mètode `actionReset:` reseteja els moviments del "Cub" deixant-lo en l'estat inicial, per a que l'usuari pugui començar a manipular-lo des de zero.

El mètode `actionShuffle:` barreja el "Cub" de forma exhaustiva tenint en compte el nombre de plans, talls i peces que el formen. La barreja és aleatòria, tant en la selecció del pla, del nivell així com de l'angle.

En futures millores, poden afegir-s'hi accions per cancel·lar moviments, reordenació manual de les peces, o accions específiques dels modes de Modelat i Disseny.

6.4.3 Control de l'Aparença

El controlador de Rubik com la majoria de controladors controla aspectes de l'aparença, com per exemple el canvi d'orientació de l'aparell. El simulador acostuma a utilitzar-se en horitzontal però interessa que la barra d'eines ocupi el mínim d'espai de la pantalla i per això el controlador només permet la posició vertical de l'aparell.

```
- (BOOL) shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)tInterfaceOrientation;
- (BOOL) shouldAutorotate;
- (NSUInteger) supportedInterfaceOrientations;
- (UIInterfaceOrientation) preferredInterfaceOrientationForPresentation;
```

Altres aspectes que ha de controlar és quan la vista apareix de nou, o quan canvia de mida, implementant els dos mètodes següents. Això passa quan s'ha seleccionat un "Cub" diferent des del llistat de l'aplicació, o en els nous iPhones que tenen una pantalla més llarga que el sistema n'estira la vista. En ambdós casos, crida al mètode `layoutRubik` que invalida les seleccions i torna a dibuixar l'escena en la vista.

```
- (void) viewWillAppear:(BOOL)tAnimated;
- (void) viewDidLayoutSubviews;
```

6.4.4 Dibuixat i Seleccionat

El controlador també té els següents mètodes per dibuixar i seleccionar l'escena els quals seran cridats quan hi hagi un moviment del "Cub" o del punt de vista d'aquest.

```
- (void) selectRubik;
- (void) drawRubikCut:(RBK9Cut*)tCut rotate:(RBK9Matrix*)tRotate;
```

El mètode `selectRubik` bàsicament s'encarrega que la vista dibuixi la selecció i la desi al `_selectBuffer`. Com que aquest mètode només s'executa després d'un moviment, automàticament al final ja s'encarrega de configurar la vista per al dibuixat (`configDraw`) i establint la textura (`applyTexture:`) per estalviar temps en el pròxim dibuixat.

El mètode `drawRubikCut:rotate:` s'encarrega de dibuixar el "Cub" amb les peces de sobre el tall (`tCut`) girades amb la matriu de rotació `tRotate`, mentre les altres peces queden sense girar. Si no s'especificat alguna de les dues variables es dibuixen totes les peces en la seva posició actual.

6.4.5 Moviment de Peces

Els moviments de les peces es podrien fer instantàniament, però això en dificultaria l'experiència a l'usuari, i per tant cal animar-les d'alguna manera. D'això se n'encarreguen els següents mètodes.

```
- (void) animateRotationLevel:(NSUInteger)tLevel
                      ofPlane:(RBK9Plane*)tPlane
                       toTurn:(float)tTurn
                    increment:(float)tIncrement;
- (void) timerRotate:(NSTimer*)tTimer;
```

El primer mètode es crida quan es vol fer un moviment d'un nivell del "Cub". Aquest mètode simplement activa el temporitzador `_timerRotate` amb la informació necessària per realitzar l'animació (pla de gir, nivell, interval de gir, increment i valor inicial), cridant al mètode `timerRotate:` a intervals definits.

A cada interval el mètode `timerRotatte:` n'extreu la informació per trobar-ne el nivell sobre el que es volen girar les peces, i un cop incrementat el valor de gir, re-dibuixar el "Cub" amb el gir calculat, cridant al mètode `drawRubikCut:rotate:` amb el tall que indica les peces a girar i la matriu de rotació amb l'angle de gir. Si l'increment ha superat el límit del moviment, el mètode aplica el gir sobre les peces del "Cub" i el dibuixa, prèviament invalidant la selecció i aturant el temporitzador.

6.4.6 Delegació de la Vista

Per a que el controlador pugui manipular el "Cub" a partir dels gestos de l'usuari sobre la vista cal que adopti el protocol `RBK9RubikViewDelegate` i implementar els següents mètodes que es criden a l'inici, al canvi, i al final; dels gestos per moure i girar la càmera i per seleccionar les peces.

Girar la càmera

Quan l'usuari selecciona amb dos dits sobre la vista i els gira sobre el seu centre, la vista `RBK9RubikView` s'encarrega d'informar al seu delegat que l'usuari vol girar la càmera, cridant els següents mètodes.

```
- (void) rubikViewWillBeginRotateCamera:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didRotateCameraRadians:(float)tRadians;
- (void) rubikView:(RBK9RubikView*)tView didEndRotateCameraRadians:(float)tRadians;
```

El primer mètode es crida quan comença el moviment de rotació, i per defecte no fa res. Podria no estar implementat.

El segon mètode es crida quan la rotació canvia indicant en radiants el gir que es vol fer. La implementació d'aquest mètode s'encarrega de modificar la rotació de la càmera, re-dibuixar l'escena i tornar a recuperar la rotació anterior.

El tercer mètode es crida quan la rotació ha finalitzat indicant el radiants de rotació a fer sobre la càmera. La implementació d'aquest mètode modifica la rotació de la càmera, en re-dibuixa l'escena i n'invalida la selecció.

Moure la càmera

Quan l'usuari desplaça dos dits sobre la vista `RBK9RubikView`, aquesta s'encarrega d'informar al seu delegat que l'usuari vol moure la càmera en la direcció del punt indicat, cridant el següents mètodes.

```
- (void) rubikViewWillBeginMoveCamera:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didMoveCameraVector:(CGPoint)tVector;
- (void) rubikView:(RBK9RubikView*)tView didEndMoveCameraVector:(CGPoint)tVector;
```

El primer mètode es crida quan comença el moviment de la càmera, i per defecte no fa res.

El segon mètode es crida quan el mou la càmera. La seva implementació primer guarda la posició anterior de la càmera, en calcula la nova posició depenent de moviment realitzat, re-dibuixa l'escena amb la nova càmera, i finalment torna a recuperar la posició anterior per a que els moviments no siguin acumulatius.

L'últim mètode es crida quan el moviment de la càmera ha finalitzat. La seva implementació calcula la nova posició de la càmera, en re-dibuixa l'escena i n'invalida la selecció.

Moure les peces

Quan l'usuari desplaça un dit sobre la vista `RBK9RubikView`, aquesta s'encarrega d'informar al seu delegat que l'usuari vol moure les peces indicant en cada moment quina peça i cada ha seleccionat, cridant el següents mètodes.

```
- (void) rubikViewWillBeginSelect:(RBK9RubikView*)tView;
- (void) rubikView:(RBK9RubikView*)tView didSelectPiece:(NSUInteger)tPiece
                                         andFace:(NSUInteger)tFace;
- (void) rubikView:(RBK9RubikView*)tView didEndSelectPiece:(NSUInteger)tPiece
                                            andFace:(NSUInteger)tFace;
```

El primer mètode es crida al començament d'intentar moure les peces. La seva implementació comprova que el buffer de selecció de la vista sigui valid cridant el mètode de la vista `selectIsValid`, i si no ho és el calcula cridant al mètode del controlador `selectRubik`. També s'encarrega de buidar els arrays de peces i cares seleccionades.

El segon mètode es crida quan se seleccionen peces indicant el número de peça seleccionada i el numero de la seva cara. La implementació d'aquest mètode s'encarrega d'afegir la nova peça i cara a l'array corresponent si aquesta peça no s'havia seleccionat prèviament.

A continuació quan ja té un mínim de dues peces intersecciona els nivells de cada peça per veure quants n'hi ha en comú. Quan només queda un nivell en comú es considera que l'usuari vol moure aquelles peces.

Un cop trobat el nivell que es vol girar, cal calcular el sentit del gir. Això es fa calculant la suma de la normal del pla de gir; i la normal del pla definit per l'origen de coordenades, el primer vèrtex de la cara de primera peça seleccionada i el primer vèrtex de la cara de la segona peça seleccionada. Si la suma té una distància inferior a $\sqrt{2}$ el sentit és negatiu (sentit horari), i si és superior o igual el sentit és positiu (sentit trigonomètric).

Amb totes les dades conegudes (nivell, pla, sentit) es crida al mètode `animateRotationLevel: ofPlane: toTurn: increment:` per realitzar l'animació de moviment.

L'últim mètode es crida quan s'ha acabat l'intent de moure peces. La seva implementació simplement buida els arrays de peces i cares seleccionades prèviament.

Capítol 7

Conclusió

Aquest capítol conclou la memòria del projecte explicant les línies obertes que podrien ampliar el projecte, així com les conclusions que se n'han obtingut.

7.1 Línies Obertes

Com s'ha comentat en diversos capítols, el projecte del simulador s'ha limitat al 'Mode de Joc' que permet a l'usuari Jugador manipular els diferents "Cubs de Rubik", o dit planament jugar-hi.

Això obre un seguit de línies tant en l'aspecte del simulador, com en el de l'aplicació que l'utilitza i s'enumeren a continuació. Algunes de les quals s'han analitzat en l'Apèndix B.

7.1.1 Simulador

A banda del 'Mode de Joc' una opció molt interessant seria que el simulador servís també per dissenyar nous "Cubs de Rubik". Per a que això fos possible s'hauria d'implementar altres modes de funcionament.

Mode de Modelat

El 'Mode de Modelat' serviria per poder crear la geometria exterior del nou "Cub". Bàsicament seria un editor en 3D simple que permetria crear figures bàsiques com cubs, cilindres, esferes, plans, etc. i una sèrie d'operacions d'unió, divisió i intersecció per a poder manipular-les

Mode de Pintat

Per a que el nou "Cub" tingués un aspecte presentable, caldria tenir el 'Mode de Pintat' per poder donar-li color i textura a les superfícies de la geometria modelada. Aquest mode hauria de permetre assignar un material o una textura a cadascuna de les cares de l'objecte, o fins i tot permetre dibuixar-hi a sobre com si fos un llenç.

Mode de Tallat

Per últim s'hauria d'implementar un 'Mode de Tallat' per dissenyar els talls de la geometria que generen les peces i estableixen els moviments possible. El resultat seria un nou "Cub de Rubik" funcional.

Mode de Configuració

Un altre mode interessant és el que permetria a l'usuari desplaçar les peces a diferents posicions del "Cub" per a part d'una posició coneguda. Així es podria ajudar a solucionar "Cubs" amb configuracions conegudes.

Moviments Locals

Per poder ajudar a l'usuari a resoldre un "Cub" és molt interessant poder identificar els diferents nivells del "Cub" en relació a la posició d'aquest, és a dir a com el veu. O com l'ha de moure per poder començar a fer una sèrie de moviments que el duguin a solucionar-lo.

Això és veu molt clarament els el Cub de Rubik original on la majoria de les instruccions per resoldre'l estan basades en la posició del Cub anomenant a les cares depenent de la seva posició: dreta, esquerra, sobre, sota, davant i darrera.

Detecció de la Solució

També seria interessant que el simulador pogués detectar automàticament la solució del "Cub" per poder felicitar a l'usuari, així com per tenir un registre dels moviments o el temps que li ha calgut per resoldre'l.

7.1.2 Aplicació

Amb les millores del simulador, a l'aplicació se li obren un gran ventall de possibilitats que podrien implementar-s'hi.

Afegir, esborrar, enviar

Una afegit que no costaria gaire seria permetre afegir nous "Cubs" al llistat de l'aplicació, així com eliminar-los. També podria ser interessant enviar-los a altres usuaris.

Modes afegits

Des de l'edició del llistat de "Cubs de Rubik" prement sobre cadascun es podria oferir la possibilitat d'entrar en els modes de disseny del "Cub" enumerats en el punt anterior: modelat, pintat i tallat.

Puntuació i Joc

Amb la possibiltiat de saber quan el "Cub" està solucionat, una opció interessant a afegir-hi seria integrar-hi un sistema de puntuació i joc col·lectiu integrat amb el Game Center d'Apple.

Registre de Moviments

Així com en els escacs els jugadors registren els moviments de cada peça i jugador, també podria ferse amb els moviments de les peces del "Cub", permeten a l'usuari tornar endavant i enrere per poder analitzar com li ha funcionat.

Algorismes de Resolució

No seria fàcil, ja que hi ha molta varietat de "Cubs de Rubik" i de diverses dificultats, però no s'ha de desestimar l'opció d'oferir algorismes de resolució per a cada tipus de "Cub". Aquest algorismes de resolució podrien integrar-se perfectament amb el registre de moviments per guiar a l'usuari en la seva resolució.

7.2 Conclusions

Després de setmanes de treball més o menys constant, s'ha arribat al final del Treball Final de Carrera que es resumeix en la present memòria i el la presentació adjunta. Treball on s'ha estudiat, dissenyat i implementat el nucli principal d'un simulador de "Cubs de Rubik" independent de la geometria, del nombre de peces, així com dels moviments entre elles.

La principal tasca que s'ha hagut de superar ha estat estudiar l'anomenat 'Model de Negoci' del projecte, que era conèixer i identificar els elements que intervenen en un "Cub de Rubik", les seves relacions i el funcionament d'aquests elements per a que fos possible dissenyar qualsevol "Cub de Rubik". Tasca que s'ha resolt amb èxit per al propòsit que es cercava, però que haurà de fer-se amb més rigor si es volen crear algorismes de resolució.

Un cop estudiat el funcionament dels "Cubs de Rubik" s'han dissenyat les classes del model que el simula fent que siguin un nucli tancat i auto-gestionat. Model que només ha variat en la fase d'implementació en el nom de les classes per a que fossin més fàcils de relacionar amb la tasca que realitzaven. El model ha estat testejat exhaustivament, classe per classe, en cadascuna de les seves funcionalitats, comprovant-ne el bon funcionament tant en els casos previstos i coneguts, com el casos aleatoris per comprovar-ne la robustesa en cada compilació.

S'ha tingut l'oportunitat d'introduir-nos en l'estudi i utilització de les llibreries OpenGL, per a poder representar els "Cubs de Rubik" en 3D. No obstant no hi ha hagut prou temps per utilitzar la versió 2.0 de l'OpenGL, que tot i ser més potent, deixa en mans del programador el càlcul de la il·luminació, dels materials i de les textures.

Tot i que sóc una persona poc organitzada, la planificació ha fet que s'hagin assolit amb èxit les fites programades, i que el projecte hagi arribat al final sense deixar caps per lligar.

Tot i l'estrès intrínsec d'estar fent un Treball Final de Carrera, he gaudit molt tornat a desenvolupar un projecte amb aquell punt d'interès i repte que et fa sentir viu.

Apèndix A

Captures de l'aplicació

Aquest apèndix mostra les captures de l'aplicació entregada, a falta de poder-les gravar en vídeo.

A.1 Arrancada del Programa

Quan arranca el programa l'aplicació de forma animada presenta la imatge inicial que en aquest cas és un llistat en blanc. Un cop acabada l'animació l'aplicació carrega els cubs i els visualitza en el llistat.

Figura A.1: Captura de l'arrancada del programa

A.2 Llistat de "Cubs"

Quan tenim el llistat de "Cubs de Rubik" podem manipular-los prement el botó d'editar, fent que la barra d'eines hi apareguin nous botons i cada cub del llistat hi surti un botó vermell que permet esborrar-lo. Si es prem algun d'aquests botons vermells hi apareix a la seva dreta un altre botó per a confirmar-ne l'esborrat, que es cancel·la si es prem en qualsevol altre lloc. Fent clic sobre el botó Done, se surt del mode d'edició.

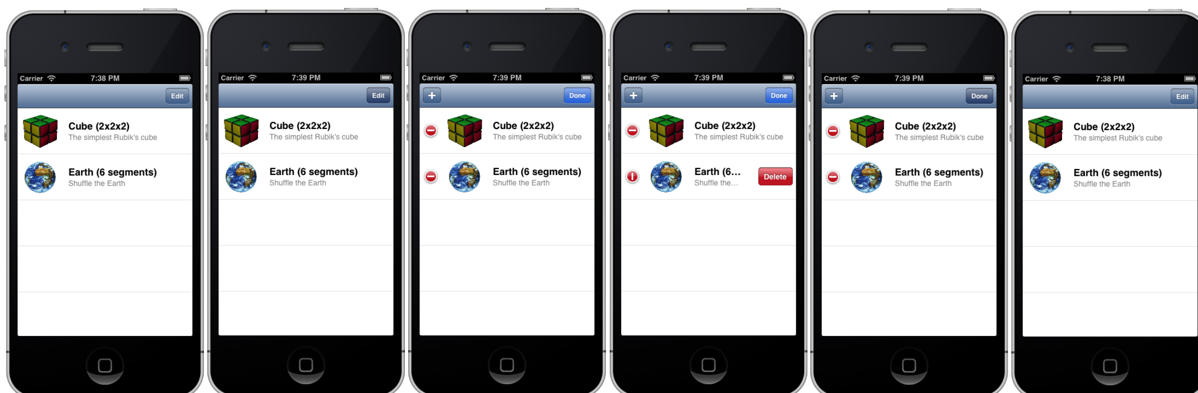

Figura A.2: Captura jugant amb el llistat

Quan estiguin implementats els models de modelat i disseny, s'hi accedirà des del llistat en mode edició prement sobre cadascun dels "Cubs". I amb el botó '+' es podrà afegir crear i copiar nous "Cubs".

A.3 Seleccionar un "Cub"

Per poder simular (jugar) amb un "Cub" simplement cal prémer sobre seu i automàticament es carrega el cub i es mostra en la pantalla del simulador realitzant una transició cap a l'esquerra.

Figura A.3: Captura al seleccionar un "Cub"

A.4 Rotació de la càmera

Seleccionant amb dos dits sobre el "Cub" (al simulador prement la tecla ALT) i girant-los respecte el seu centre, el simulador gira la càmera fent l'efecte que el "Cub" volta sobre el seu centre..

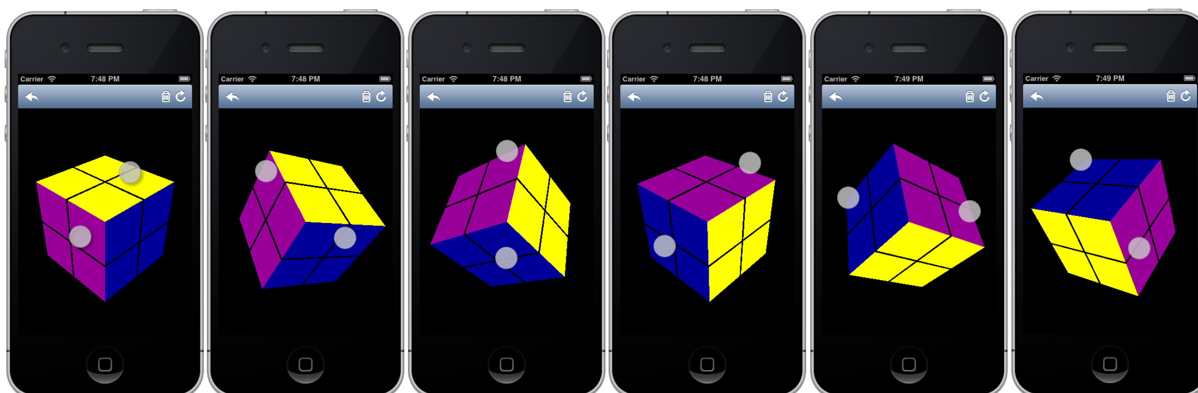

Figura A.4: Captura al girar la càmera

A.5 Moure la càmera

Seleccionant amb dos dits sobre el "Cub" (al simulador prement les tecles ALT + SHIFT) i desplaçant-los cap a un costat o amunt i avall, la càmera es mou al voltat del "Cub" fent l'efecte que el girem.

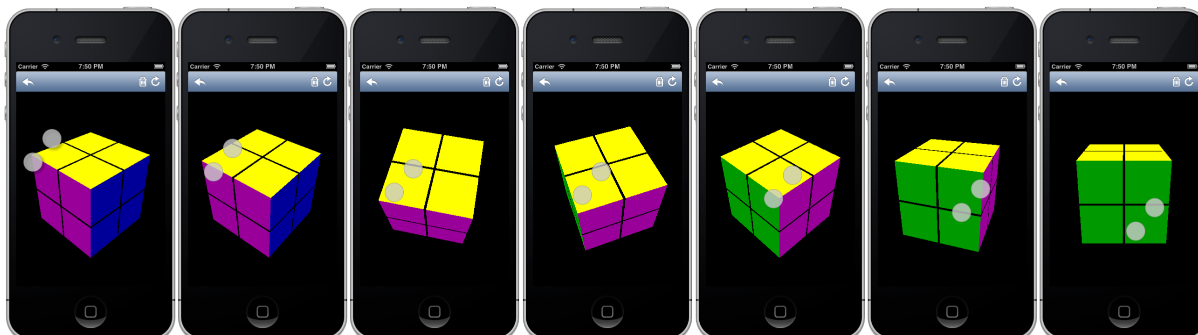

Figura A.5: Captura al moure la càmera

A.6 Movent les peces

Per moure les peces cal seleccionar-ne un mínim de tres (en el "Cub" que es mostra en les captures). Això es fa seleccionant sobre una de les peces i recorrent per sobre les les altres dues de forma circular en la direcció que es vulgui fer el moviment.

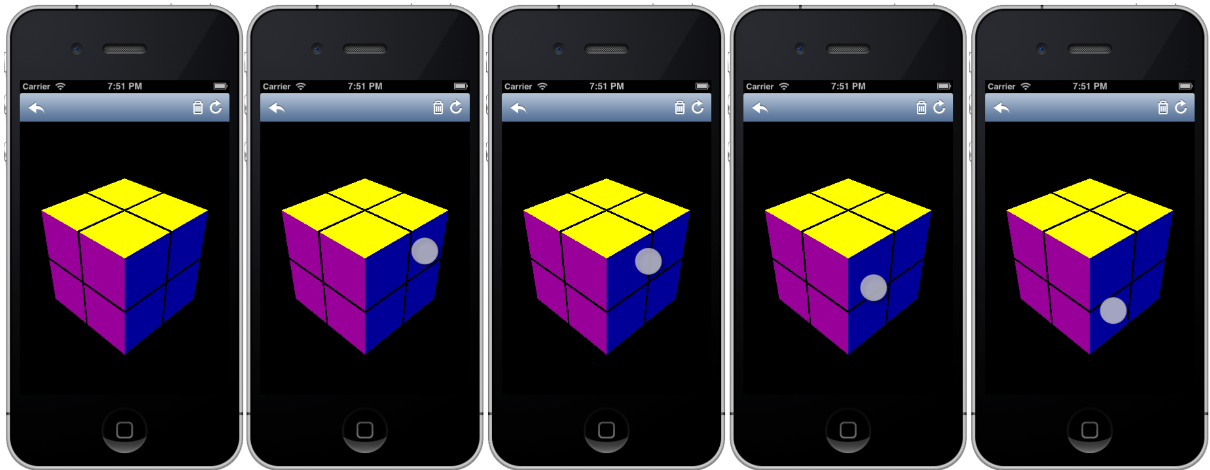

Figura A.6: Captura del gest de moure peces

Un cop s'ha tocat tres peces d'un mateix nivell, el simulador en gira totes les peces que hi pertanyen en una suau animació.

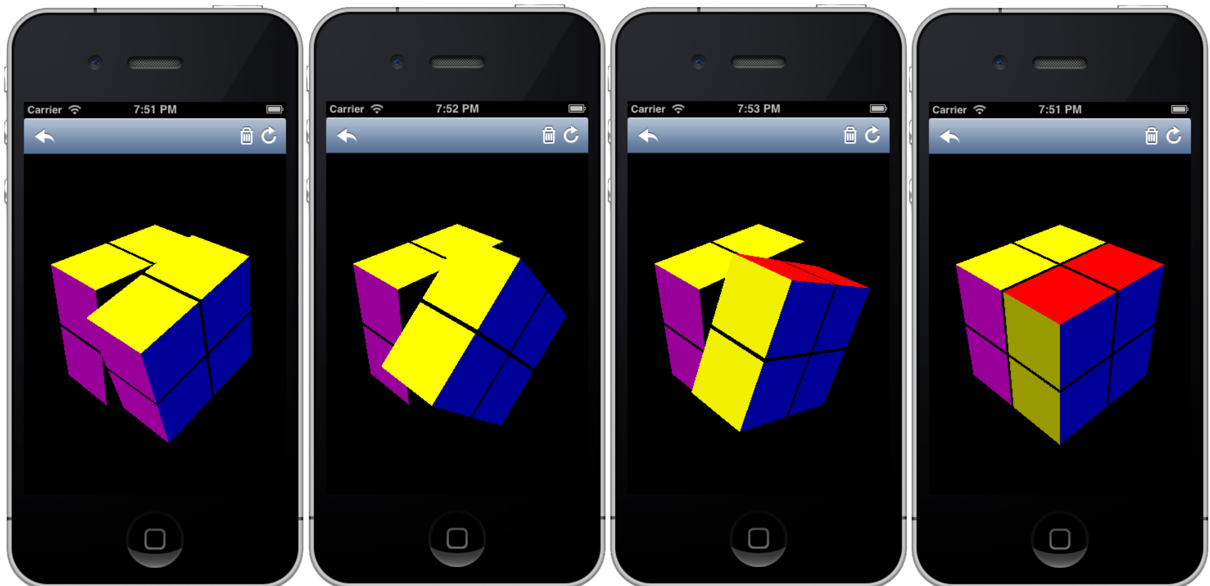

Figura A.7: Captures de l'animació del moviment

Apèndix B

Altres Modes

Aquest capítol se centra en el disseny del simulador tant des del punt de vista funcional com el de les classes. També es descriu en la secció de prototipatge les interfícies d'usuari amb les seves respectives pantalles. Per acabar es dissenya la validació amb usuaris dels dissenys proposats en aquest mateix capítol.

S'ha estructurat el capítol en les següents seccions:

- **Funcional**: On es realitza el disseny funcional, amb la identificació dels tipus d'usuari i dels cassos d'ús.
- **Prototipatge**: On es mostra el disseny de les pantalles de la interfície gràfica.

B.1 Usuaris

Tenint en ment el conjunt de funcionalitats que pot tenir el simulador en un futur, s'han establert tres tipus d'usuari a considerar en l'ús del simulador:

- **Modelador**: Aquest usuari s'encarrega de modelar les figures i peces que formaran el cub.
- **Dissenyador**: És l'usuari encarregat de dissenyar les parts mòbils del "cub" a partir de la figura modelada.

A partir de cada tipus d'usuari, es presenten els seus casos d'ús, juntament amb una breu descripció de cadascun d'ells.

El lector pot observar que alguns casos d'ús estan tintats de color vermell i d'altres no. Els que no ho estan són els casos i mòduls que s'han d'implementar en el projecte. En canvi els casos i mòduls tintats, quedaran especificats i dissenyats però no necessàriament implementats.

B.1.1 Modelador

Els primers cassos d'ús estan centrats en l'usuari modelador, què s'encarrega de modelar les figures de les quals es crearan els "cubs". El modelador tracta amb els següents quadre mòduls.

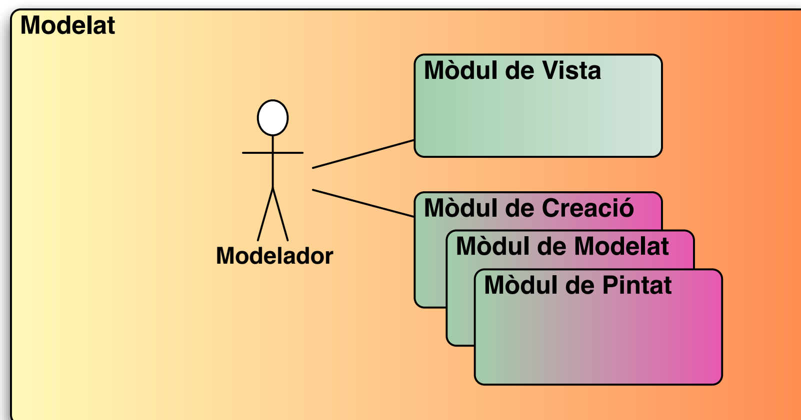

Figura B.1: Cassos d'Ús — Modelador

- **Mòdul de Vista**, què s'encarrega de manipular el punt de vista des d'on l'usuari veu el model en que treballa.
- **Mòdul de Creació**, què s'encarrega d'afegir figures, moure-les i eliminar-les.
- **Mòdul de Modelat**, què s'encarrega de manipular les figures: unint-les, tallant-les, ...
- **Mòdul de Pintat**, què s'encarrega de pintar la superfície de la figura.

B.1.2 Dissenyador

Els cassos d'ús del dissenyador s'encarreguen de tallar la figura en diferents parts per dotar de funcionalitat al "cub". Aquest tracta amb quadre mòduls.

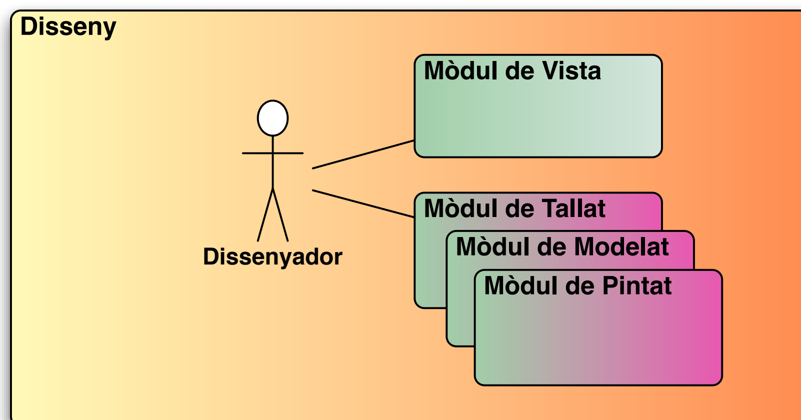

Figura B.2: Cassos d'Ús — Dissenyador

- **Mòdul de Vista**, què s'encarrega de manipular el punt de vista des d'on l'usuari veu el model en que treballa.
- **Mòdul de Tallat**, què s'encarrega de fer els talls a la figura per generar el "cub".,
- **Mòdul de Modelat**, què permet modelar cada peça que s'hagi generat en el tallat.
- **Mòdul de Pintat**, què permet pintar la superfície de cada peça que s'hagi generat en el tallat.

B.2 Casos d'Ús

A continuació es presenten amb més detall els mòduls previstos per al simulador, essent els principals: el mòdul de vista i el de moviments. La resta tot i que s'han dissenyat, difícilment podran implementar-se degut a la seva complexitat i el temps disponible.

Cal afegir, també, que no s'han considerat els cassos de desar i carregar continguts per treballar amb el simulador, ja que es pressuposen. Així mateix la descripció de cada cas d'ús serà superficial ja que s'han analitzat amb més detall en el capítol d'Anàlisi.

Mòdul de Creació

El mòdul de creació ha de permetre afegir elements geomètrics per crear la figura que servirà de base per crear el "cub".

El primer cas d'ús que s'hi pot veure ha de permetre crear i afegir elements tridimensionals, ja siguin volums o superfícies. El segon dels casos, ha de permetre modificar-ne la posició i orientació dels elements existents, així com les seves dimensions. Per acabar, hi ha la possibilitat d'eliminar-ne els elements.

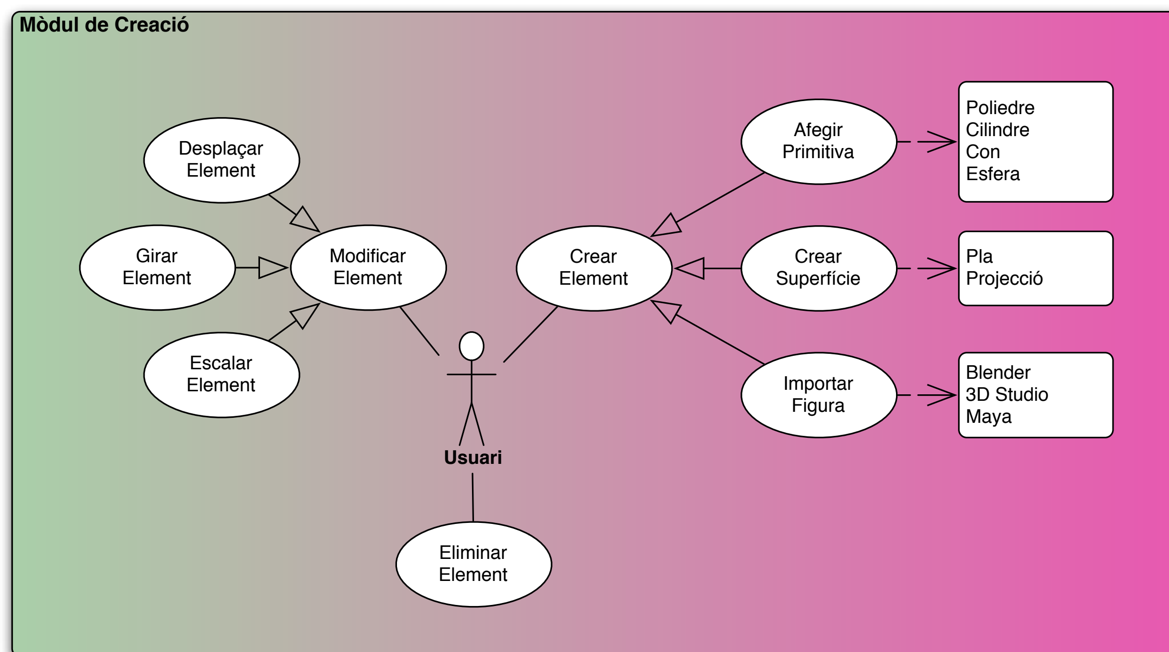

Figura B.3: Cas d'Ús — Mòdul de Creació

B.2.1 Mòdul de Modelat

El mòdul de modelat, ha de permetre modelar les figures tridimensionals, per aconseguir figures més complexes.

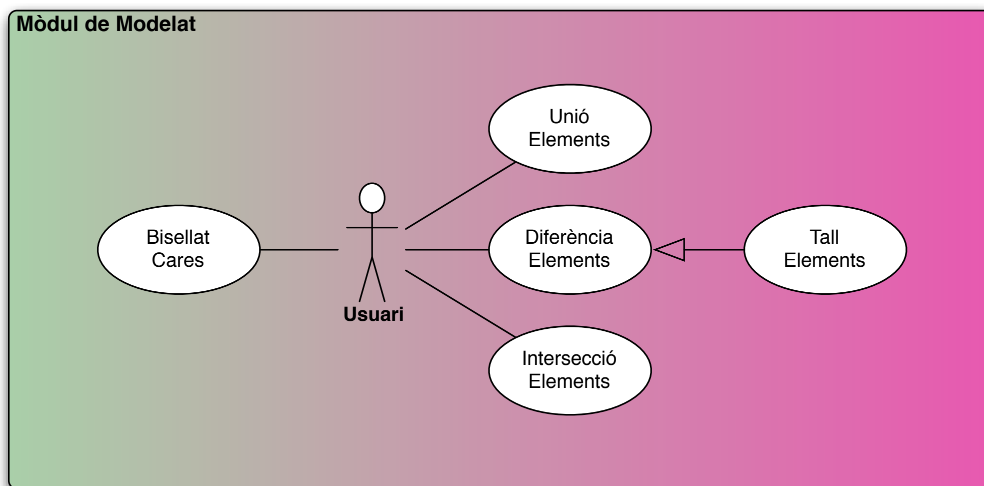

Figura B.4: Cas d'Ús — Mòdul de Modelat

Els casos d'ús que s'han identificat són els típics de programes de disseny tridimensional. Hi ha la unió de d'elements, la diferència i la intersecció. A partir de la diferència se n'ha estès el tall d'un element en dos trossos. També s'ha cregut oportú permetre definir un bisellat al voltant d'una cara.

Altres casos d'ús s'hi poden afegir per augmentar-ne la funcionalitat.

B.2.2 Mòdul de Pintat

El mòdul de pintat ha de permetre assignar color o textura a les cares d'una figura.

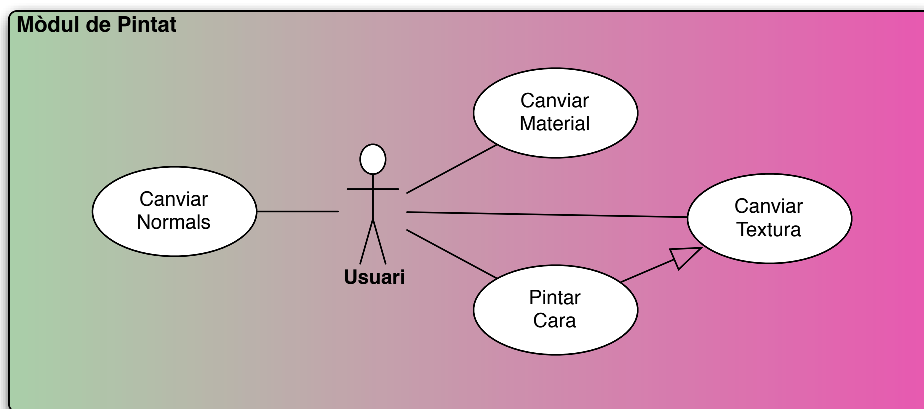

Figura B.5: Cas d'Ús — Mòdul de Pintat

El cas de canviar les normals serveix en el cas que s'hi vulgui afegir il·luminació a l'escena i simular que una superfície és arrodonida. El cas de canviar de material permet definir els colors del material a una superfície. Els altres dos casos de pintat es basen en l'extensió d'aplicar una imatge de textura sobre la cara. Mentre canviar la textura situa una imatge sobre la superfície, la de pintat la cara n'és una especialització que permet a l'usuari pintar la imatge que s'utilitzarà com a textura.

B.2.3 Mòdul de Tallat

El mòdul de tallat s'encarrega de generar el "cub" dividint la figura en les peces que la formaran.

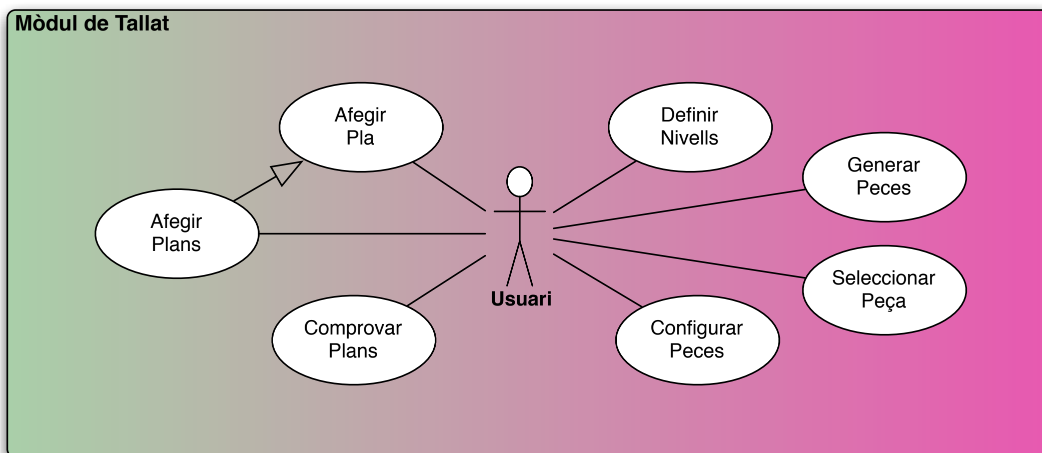

Figura B.6: Cas d'Ús — Mòdul de Tallat

S'hi poden veure els casos d'ús per afegir-hi plans de tall, així com el que en comprova la validesa. També hi ha el cas que permet definir els nivells en que es divideix la figura base, obtenint les peces en cada pla de tall. Per acabar es generen les peces, i es permet configurar-ne les relacions per si cal considerar que dues peces són iguals, o si alguna té algun tipus de simetria.

B.2.4 Mòdul de Peces

L'últim mòdul, el de les peces, hauria de permetre al jugador re-situar les peces dins la figura en la posició que cregui convenient sempre que sigui vàlida.

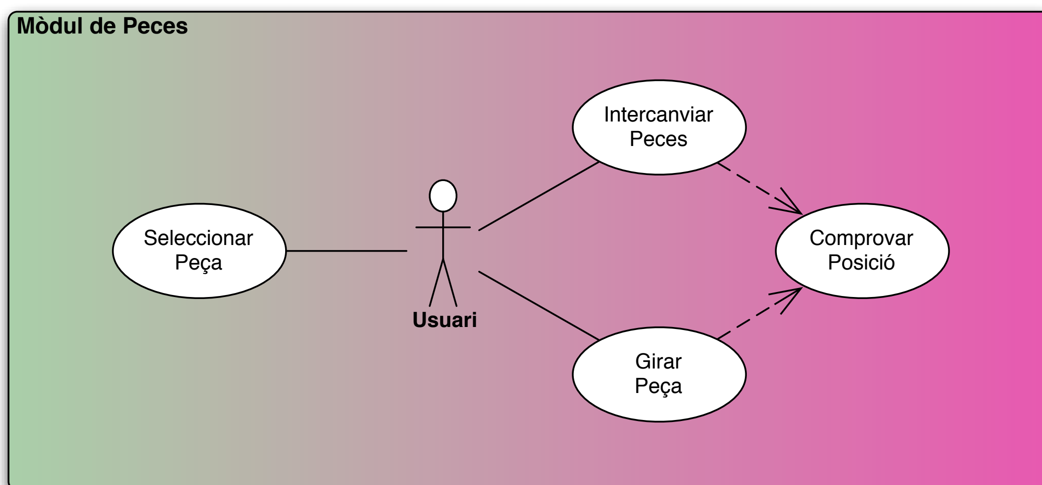

Figura B.7: Cas d'Ús — Mòdul de Peces

El primer cas que s'hi veu és el que permet seleccionar una peça, molt útil si s'interactua visualment. Els altres dos casos permet intercanviar dues peces —comprovant-ne que poden estar en la mateixa posició— i l'últim per girar la peça en la posició actual.

B.3 Prototipat

En aquesta secció es presenta l'estructura de pantalles que disposaria una aplicació tipus que pogués interactuar amb el simulador tant en la creació i disseny dels "cubs", com en el joc.

No obstant això, la implementació se centrarà en la simple simulació/joc, degut a la complexitat de desenvolupar les pantalles d'edició amb el poc temps disponible.

B.3.1 Pantalla de Disseny

Estant a la pantalla principal amb l'opció d'editar activada es pot prémer sobre un "cub" i s'entra en la pantalla d'edició. Aquesta pantalla és igual que la de joc, amb l'única diferència que la barra d'eines s'adapta a cada funcionalitat.

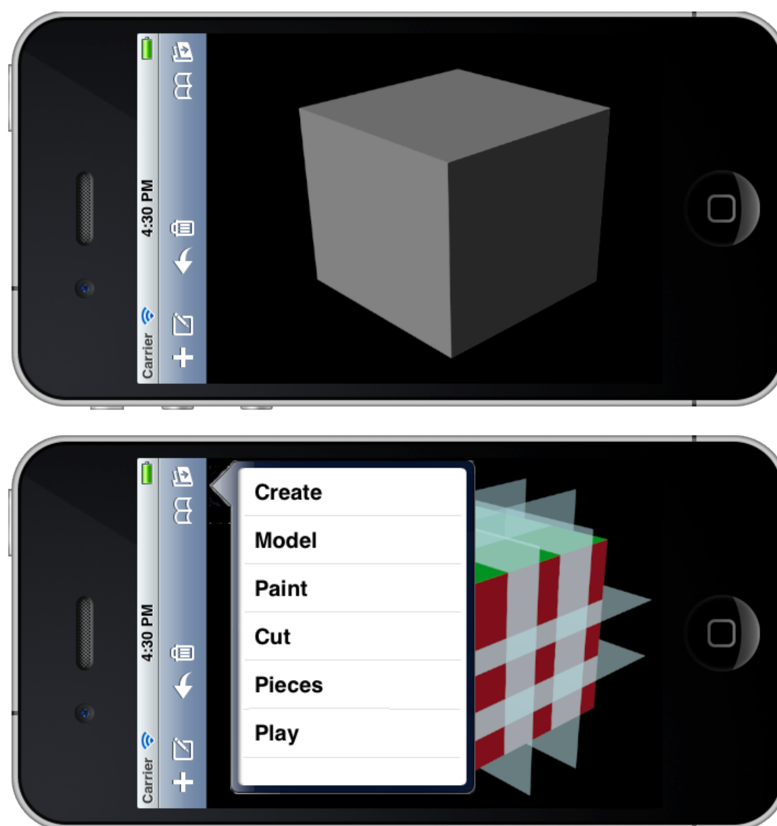

Figura B.8: Pantalla d'Edició — Dibuixat — Canvi de Mode

L'edició del cub funciona en diferents modes depenent del que es vulgui fer: el dibuixat, el modelat, el pintat, el tallat, la configuració i la simulació; que es descriuen a continuació.

Dibuixat

El mode de dibuixat permet afegir només figures en tres dimensions: cubs, tetraedres, cilindres, esferes, plans, etc. També permet seleccionar-los, desplaçar-los, girar-los i escalar-los.

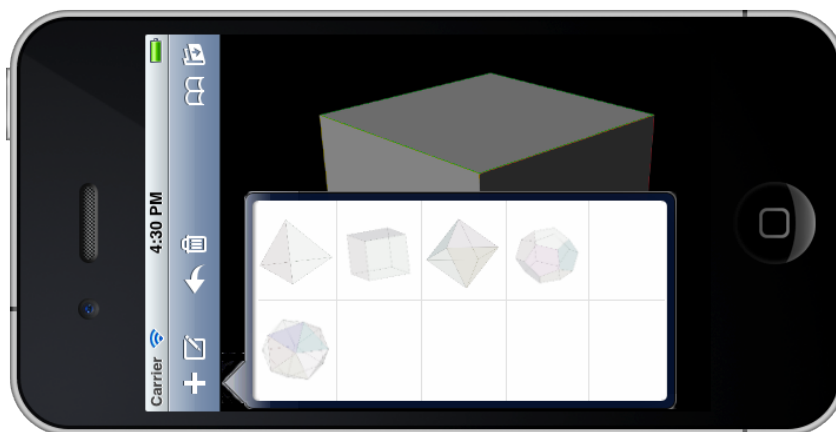

Figura B.9: Pantalla d'Edició — Mode de Dibuixat

Modelat

El mode de modelat permet realitzar operacions entre els diferents elements dibuixats. Algunes de les operacios possibles serien: unió, diferència i intersecció.

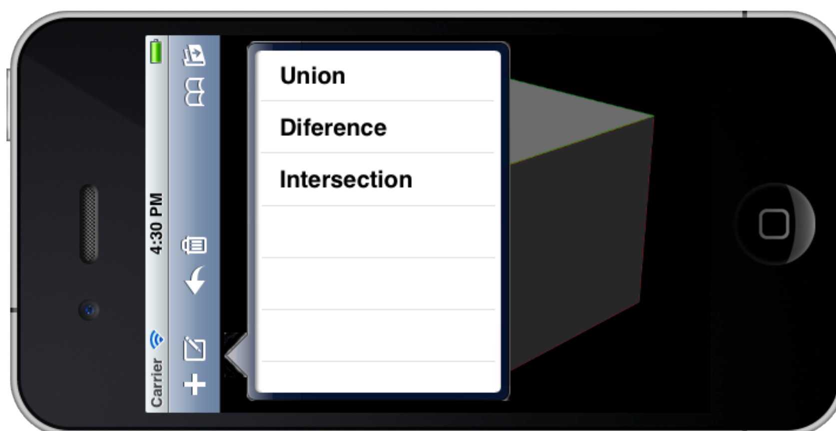

Figura B.10: Pantalla d'Edició — Mode de Modelat

Pintat

El mode de pintat permet assignar color a cada cara de la figura. També es preveu aplicar-hi textures o pintar-hi directament com si fos un llenç.

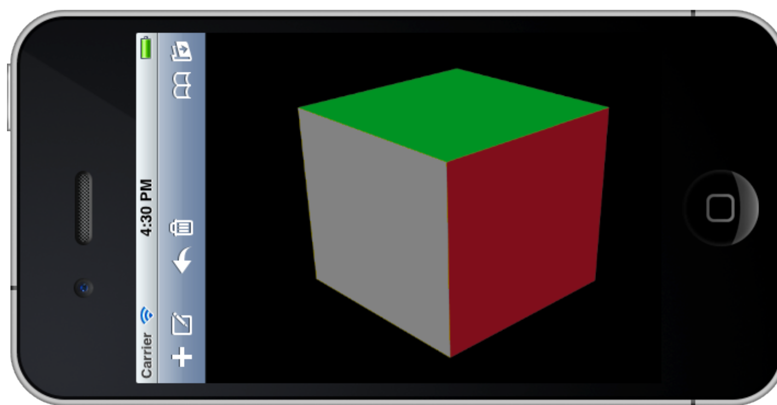

Figura B.11: Pantalla d'Edició — Mode de Pintat

Tallat

Un cop es té la figura base realitzada i pintada es pot passar al mode de tallat en que es configura el "cub" dividint-lo en les diverses peces que el formen i definit-ne els moviments que s'hi podran aplicar.

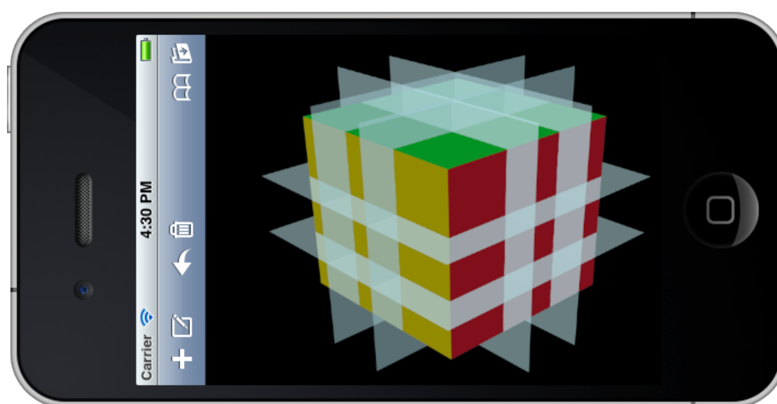

Figura B.12: Pantalla d'Edició — Mode de Tallat

Test

Per últim hi ha el mode de test que permet simular el funcionament del cub. És idèntic que el mode de Joc descrit anteriorment.

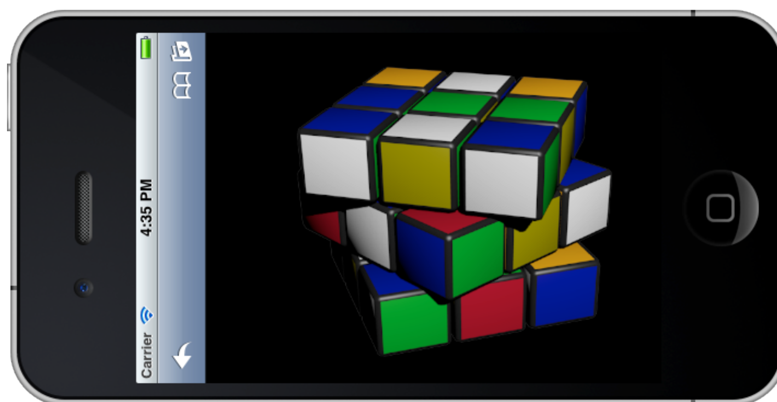

Figura B.13: Pantalla d'Edició — Mode de Test

Apèndix C

Avaluació de Prototipus

Aquest apèndix presenta la planificació de l'avaluació de prototipus.

C.1 Estratificació

Per a conèixer els diferents perfils d'usuaris que realitzaran les proves d'avaluació de prototipus, cal realitzar-los-hi una conjunt de preguntes que puguin estratificar aquests usuaris en una sèrie de perfils desitjats.

C.1.1 Perfils

Els perfils que es vol aconseguir estratificar estan agrupats en els següents aspectes:

- **Sexe**: El sexe és un dels aspectes més importants a l'hora de generar perfils en el sector del joc.
- **Edat**: Tot i que l'edat en sí no és cap aspecte imprescindible, permet agrupar els usuaris per generacions i per adopció tecnològica.
- **Jugador**: A l'estar avaluant una aplicació dirigit bàsicament a jugadors, és important estratificar la seva opinió respecte les seves preferències, per veure si s'aconsegueixen les expectatives d'uns i si s'incorporen nous jugadors.
- **Formació**: El Cub de Rubik és un trenca-closques d'un nivell alt de dificultat, però de la forma que es presenta, permetent diverses configuracions, cal veure si convenç a qui els hi fa respecte i manté l'interès dels qui cerquen nous reptes.
- **Tecnologia**: Cal també veure com varien les respostes dels usuaris depenent són nous usuaris de tecnologies tàctils, o ja fa temps que les han adoptat.

No s'ha considerat en cap cas la cultura o la procedència, ja que l'aplicació va destinada a gent amb una cultura tecnològica comuna.

C.1.2 Preguntes

A continuació es mostren les preguntes que serviran per estratificar els diferents usuaris:

- Quina edat teniu?

- Sou home o dona?
- Sou jugador habitual, ocasional o no ho sou?
- Jugueu principalment a jocs d'arcade, d'estratègia, de simulació, de trenca-closques, o altres?
- Useu dispositius tàctils de forma habitual, ocasional o no n'acostumeu a utilitzar?
- Teniu formació bàsica, mitja o superior?

C.2 Tasques

Per avaluar els prototipus, els usuaris hauran de fer una sèrie de tasques destinades a avaluar cada aspecte del disseny del prototipus.

C.2.1 Aspectes a valorar

És important segmentar els aspectes a valorar per procurar identificar correctament les causes i els efectes sobre els quals s'hi hauran de centrar els esforços. Els aspectes que s'ha cregut adient de valorar són els següents:

- La **Presentació** ha de valorar la impressió dels usuaris a l'aspecte visual de l'aplicació, així com saber si han identificat correctament els elements mostrats.
- L'**Accessibilitat** ha de valorar si l'usuari pot accedir a totes les funcionalitats que s'amaguen sota la interfície, i si ho fa de forma intuïtiva.
- La **Funcionalitat** serveix per a valorar si l'usuari ha pogut realitzar les tasques amb comoditat o hi ha trobat quelcom en falta..
- La **Resposta** valorarà que l'aplicació sigui intuïtiva a l'usuari. Que faci allò que n'espera.
- La **Fluïdesa** és un aspecte més específic de la resposta que ha de valorar que l'usuari se senti còmode en el seu ús, sense haver d'esperar quan no és estrictament necessari.
- La **Satisfacció** és una valoració mitjana de la resta d'aspectes, permetent veure'n la importància que li atorga l'usuari a cadascun.

C.2.2 Tasques a realitzar

Per a valorar els diferents aspectes, se li presentaran a l'usuari una sèrie de tasques centrades en cada mòdul de l'aplicació. Tasques què haurà de realitzar en un cert temps.

- **Navegar**: Aquesta tasca està destinada a valorar la navegació entre les diferents pantalles de l'aplicació per a que l'usuari en faci una valoració introductòria, identificant-ne les funcionalitats.
- **Produir**: En aquesta tasca es procurarà que l'usuari realitzi una sèrie de tasques específiques sobre les pantalles: crear un "cub" nou, esborrar-ne un, copiar un "cub", accedir a jugar amb un "cub", accedir a l'edició d'un cub, sortir del simulador.
- **Jugar**: En aquesta tasca s'entrarà en el simulador amb un Cub de Rubik normal i s'oferirà al jugador que hi jugui. Es repetirà amb un "cub" amb una altra geometria i complexitat.
- **Girar**: Aquesta tasca servirà per veure si l'usuari ha comprés el funcionament i les possibilitats del simulador. Se li farà girar el punt de vista per veure el "cub" en una certa posició, se li farà girar certs nivells, cancel·lar un moviment, així com re-inicialitzar el "cub".

També s'han dissenyat tasques per a la funcionalitat de disseny del "cub". Aquestes però només les realitzaran els usuaris que hagin mostrat certa facilitat amb les anteriors.

- **Dissenyar**: Aquesta tasca pretén introduir a l'usuari en la pantalla d'edició de "cubs" per a que pugui veure'n el funcionament general.
- **Editar**: Un cop l'usuari ha "jugat" una mica amb l'editor se li proposaran una sèrie de tasques referents a editar un "cub", entre les quals hi haurà: Crear una figura base prefixada i acolorir-la, dissenyar els talls d'una certa manera, i generar el "cub" final.

C.3 Preguntes

Un cop els usuaris han realitzat les tasques encomanades relacionades amb el prototipus hauran de respondre a un test de preguntes relacionades amb aquestes. Per concloure es faran unes preguntes generals.

C.3.1 Navegar

- Valoreu de l'1 al 10 les següents frases, essent l'1 (molt en desacord), el 5 (d'acord) i al 10 (molt d'acord):
  - L'aspecte visual general és agradable
  - La informació mostrada és confusa
  - La navegació ha respost com esperava
- Responeu amb un Sí o un No les següents qüestions:
  - Es pot esborrar un "cub"?
  - Es pot crear un "cub"?
  - Es pot copiar un "cub"?
  - Es pot editar un "cub"?
  - Es pot jugar amb un "cub"?
- Responeu les següents qüestions:
  - Quines opcions hi heu trobat en falta?
  - Quins "cubs" del llistat no heu identificat? Per què?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.2 Produir

- Valoreu del 0 al 10 les següents frases, essent el 0 (no aconseguit), l'1 (amb dificultat), el 5 (sense dificultat) i al 10 (amb facilitat):
  - Amb quina dificultat heu entrat al simulador?
  - Amb quina dificultat heu sortit del simulador?
  - Amb quina dificultat heu afegit un "cub"?
  - Amb quina dificultat heu copiat un "cub"?
  - Amb quina dificultat heu eliminat un "cub"?
  - Amb quina dificultat heu entrat a l'editor?
- Responeu les següents qüestions:
  - Com es podrien facilitar certes tasques?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.3 Jugar

- Valoreu de l'1 al 10 les següents frases, essent l'1 (molt en desacord), el 5 (d'acord) i al 10 (molt d'acord):
  - La barra d'eines està ben situada
  - Les eines de la barra han respost com esperava
  - El moviment del cub" és intuïtiu
  - L'aspecte visual del cub és l'adient
  - L'aspecte visual de l'estrella és l'adient
  - El gir de les peces del cub és intuïtiu
  - El gir de les peces de l'estrella és intuïtiu
- Responeu amb un Sí o un No les següents qüestions:
  - Es pot moure el "cub" des de qualsevol direcció?
  - Es pot girar el "cub" a dreta i esquerra?
  - Es poden girar un grup de peces del "cub"?
  - Es poden girar més d'un grup de peces del "cub"?
  - Es pot cancel·lar l'últim gir?
  - Es pot re-iniciar el "cub"?
- Responeu les següents qüestions:
  - Quines opcions heu trobat en falta?
  - Amb quins gestos esperàveu moure el "cub"?
  - Amb quins gestos esperàveu girar les peces?
  - Amb quins gestos vos facilitaria moure el "cub"?
  - Amb quins gestos vos facilitaria girar les peces?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.4 Girar

- Valoreu del 0 al 10 les següents frases, essent el 0 (no aconseguit), l'1 (amb dificultat), el 5 (sense dificultat) i al 10 (amb facilitat):
  - Amb quina dificultat heu mogut el "cub"?
  - Amb quina dificultat heu girat un únic nivell?
  - Amb quina dificultat heu girat dos nivell alhora?
  - Amb quina dificultat heu cancel·lat l'últim gir?
  - Amb quina dificultat heu re-iniciat el "cub"?
- Responeu les següents qüestions:
  - Quines opcions hi heu trobat en falta?
  - Quins canvis gestuals creieu que en facilitaria la manipulació del "cub"?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.5 Dissenyar

- Valoreu de l'1 al 10 les següents frases, essent l'1 (molt en desacord), el 5 (d'acord) i al 10 (molt d'acord):
  - La barra d'eines està ben situada
  - El procés d'edició és intuïtiu
  - Els modes d'edició són clars
  - Les eines de modelat han respost com esperava
  - Les eines de pintat han respost com esperava
  - Les eines de tallat han respost com esperava
- Responeu amb un Sí o un No les següents qüestions:
  - Es poden afegir figures bàsiques?
  - Es poden moure, girar i escalar figures?
  - Es poden esborrar figures?
  - Es poden unir figures?
  - Es poden seccionar figures?
  - Es poden interseccionar figures?
  - Es poden pintar figures?
  - Es pot dissenyar el "cub"?
  - Es pot generar el "cub"?
  - Es pot provar el "cub"?
- Responeu les següents qüestions:
  - Quines opcions hi heu trobat en falta?
  - Com variaríeu el procés d'edició d'un "cub"?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.6 Editar

- Valoreu del 0 al 10 les següents frases, essent el 0 (no aconseguit), l'1 (amb dificultat), el 5 (sense dificultat) i al 10 (amb facilitat):
  - Amb quina dificultat heu modelat la figura?
  - Amb quina dificultat heu pintat la figura?
  - Amb quina dificultat heu dissenyat el "cub"?
  - Amb quina dificultat heu configurat el "cub"?
  - Amb quina dificultat heu generat el "cub"?
  - Amb quina dificultat heu provat el "cub"?
- Responeu les següents qüestions:
  - Com milloraríeu el modelat de la figura?
  - Com milloraríeu el pintat de la figura?
  - Com milloraríeu el disseny del "cub"?
  - Com milloraríeu el test del "cub"?
- Afegiu els comentaris addicionals que considereu oportuns

C.3.7 Conclusió

- Valoreu de l'1 al 10 les següents frases, essent l'1 (molt en desacord), el 5 (d'acord) i al 10 (molt d'acord):
  - L'experiència de joc ha estat satisfactòria
  - L'experiència de disseny ha estat satisfactòria
  - Em descarregaria aquesta aplicació
  - Utilitzaria l'aplicació freqüentment
  - Pagaria per utilitzar aquesta aplicació
- Afegiu els comentaris addicionals que considereu oportuns

Apèndix D

OpenGL

Per crear un simulador en 3 dimensions (3D), s'ha de tenir un mínim domini de la API OpenGL que permet dibuixar en 3D amb relativa facilitat, així com conèixer la forma en que la plataforma iOS l'utilitza. d'entorns en 3D.

Figura D.1: OpenGL Embedded Sistems sobre iOS

Bàsicament la API OpenGL implementa un model client-servidor proporcionant les funcions necessàries per a que l'aplicació (el client) es comuniqui amb la targeta gràfica (el servidor). Aquestes funcions permeten generar imatges.

D.1 OpenGL sobre iOS

Les plataformes mòbils —com ho és la plataforma de dispositius mòbils d'Apple, iOS— acostumen a implementar un versió "reduïda", especial per a dispositius crítics i, anomenada OpenGL ES (Embedded Sistems). Sobre iOS hi ha implementades les versions 1.1 i 2.0 per a la majoria dels dispositius més nous (a partir del 3GS), mentre que els dispositius anteriors només tenen implementats la versió 1.1.

Una característica que diferencia la implementació d'Apple sobre la resta, és que Apple no permet dibuixar directament sobre la pantalla. S'ha de fer primer sobre un buffer i després enviar-ho a la capa de la vista. Això es fa així, perquè Apple permet aplicar efectes entre la capa OpenGL i d'altres que hi puguin haver. De totes formes, no se'n recomana fer un abús en sistemes mòbils, a causa del cost computacional.

D.2 Versions d'OpenGL ES

La versió 1.1 de l'OpenGL ES proporciona una sèrie de funcions ben definides per comunicar-se amb el servidor. Aquestes funcions permeten definir una il·luminació, textura i ombrejat tradicional que es poden activar o desactivar a mesura que es van afegit els elements que ho utilitzaran.

La versió 2.0 comparteix moltes funcions de la 1.1, però elimina totes les funcions que en constrenyen la funcionalitat. En lloc seu, hi introdueix funcions que proporcionen accés a diverses etapes del càlcul. Els "shaders" són funcions implementades pel programador que s'executen directament en la targeta gràfica i que permeten personalitzar el càlcul dels vèrtexs i el pintat dels fragments, introduint-hi transformacions en les geometries i efectes de llums, ombres i textures més complexes.

La àmplia potència en la versió 2.0, queda fora de l'abast dels neòfits a l'hora de poder treballar amb llums, ombres i textures; si no coneixen els algoritmes que permeten implementar aquestes característiques dins els "shaders".

D.3 Configurant l'OpenGL ES

Per configurar OpenGL sobre la plataforma iOS s'ha de disposar primerament d'una vista amb una capa (`CALayer`) de tipus `CAEAGLLayer`. Tot i que a partir de la versió iOS 5.0, la framework ja disposa d'una vista amb aquestes característiques, es interessant implementar-la de nou per poder executar l'aplicació en sistemes amb versions anteriors.

D.3.1 Creant el context

Un cop es té la vista amb la seva `CAEAGLLayer` definida, s'ha de crear el context on l'OpenGL hi dibuixa. Això es fa en la inicialització de la vista amb el següent parell d'instruccions, indicant quina versió de la API es vol utilitzar.

```
_context = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];
[EAGLContext setCurrentContext:_context];
```

Si el `_context` no es crea amb la versió 2.0 de la API, es pot comprovar i crear-lo de nou amb la versió 1.1, sempre que l'aplicació estigui preparada per funcionar en aquesta versió.

D.3.2 Creant els buffers

Un altre tema important a l'hora de treballar amb OpenGL és configurar els buffers, els quals s'encarreguen de desar-hi informació de diferent tipus depenent del tipus de buffer. Els tipus de buffer són:

- **Frame**, què conté grups dels tres buffers següents.
- **Color**, què s'encarrega de desar la imatge final.
- **Depth**, què s'encarrega de desar la informació de profunditat dels objectes per poder-los dibuixar en l'ordre correcte.
- **Stencil**, què s'encarrega de desar la informació de les parts que s'han de pintar i les que no.

Un exemple de configuració dels buffers és el següent.

```
- (void) setupBuffers;
{
    CGRect rect = [self bounds];

    // Create frame buffers that contain other buffers
    glGenFramebuffers(1, &_frameBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, _frameBuffer);

    // Create depth render buffer to respect Z-depth information
    glGenRenderbuffers(1, &_depthBuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, _depthBuffer);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16,
                          rect.size.width, rect.size.height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER, _depthBuffer);
    glEnable(GL_DEPTH_TEST);

    // Create color render buffer to paint content
    glGenRenderbuffers(1, &_colorBuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, _colorBuffer);
    [_context renderbufferStorage:GL_RENDERBUFFER
                     fromDrawable:(CAEAGLLayer*)[self layer]];
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                              GL_RENDERBUFFER, _colorBuffer);
}
```

Les variables `_frameBuffer` i altres (de tipus buffer) es declaren com un punter amb el tipus `GLuint`. El primer que s'ha de fer és generar el Frame Buffer i assignar-lo al seu punter, per a continuació enllaçar-lo per a que l'OpenGL sàpiga quin ha d'utilitzar —sempre utilitza l'últim que s'ha enllaçat.

Seguidament es generen la resta de buffers que s'utilitzen i que s'afegiran dins del Frame Buffer. El Depth Buffer també es genera amb la mateixa instrucció i s'enllaça a l'OpenGL, després se'n configura les dimensions i el format de cada píxel; per acabar introduint-lo dins el Frame Buffer creat al principi. D'igual manera, el Color Buffer és genera, s'enllaça de forma similar, amb la diferència que el configura el propi sistema amb els seus paràmetre.

Així com el buffer de color és imprescindible per a crear una imatge amb OpenGL, la resta de buffers són opcions i s'han d'activar en cada cas amb la instrucció `glEnable(GL_DEPTH_TEST)`, amb el paràmetre adient en cada cas: `GL_DEPTH_TEST`, `GL_STENCIL_TEST`.

En les següents imatges es pot observar la diferència entre utilitzar el buffer de profunditat o no utilitzar-lo.

Figura D.2: Depth Buffer: Desactivat — Activat

D.4 Capacitats de l'OpenGL ES

En sistemes per a dispositius mòbils, com que disposen de capacitats de processament limitades, és important determinar les seves capacitats màximes, per poder adaptar-s'hi o si no és possible informar-ne a l'usuari.

Entre les capacitats a comprovar tenim:

- **Comunes**, que fan referència a la mida màxima de les textures i al nombre de profunditat de bits dels buffers.
- **Shader**, en la versió 2.0 es poden determinar les limitacions dels shaders, com: el nombre màxim d'atributs, el de uniforms, de les variacions i unitats de textures.
- **Textures**, en la versió 1.1 també hi ha limitacions en el nombre de textures i es el nombre de plans de tall.
- **Extensions**, també es pot veure quines extensions té implementades l'OpenGL.

Juntament amb determinar aquestes capacitats, és important comprovar periòdicament els errors que es vagin produint, utilitzant la funció `glGetError`. Aquesta funció té un cost de processament important, per tant s'ha d'utilitzar amb cura quan l'aplicació està en producció, i utilitzar-la més freqüentment quan s'està en desenvolupament i proves.

D.5 Tipus de dades

Una de les característiques més importants a tenir en compte a l'hora de treballar amb OpenGL és utilitzar els seus propis tipus de dades, ja que aquests són independents del maquinari i del compilador i estalvien mals de caps posteriors. Només s'enumeraran ja que qualsevol programador, a partir del seu nom, els pot identificar amb relativa facilitat,

```
GLenum, GLboolean, GLbitfield
GLbyte, GLshort, GLint
GLubyte, GLushort, GLuint
GLfloat, GLclampf
GLfixed, GLclampx
GLvoid, GLsizei
```

S'ha d'esmentar però, el fet que la majoria de funcions d'OpenGL tenen noms parametritzables depenent dels tipus de paràmetres que s'utilitzen. Aquí es mostren les abreviacions: `b` (GLbyte), `s` (GLshort), `i` (GLint), `f` (GLfloat), `ub` (GLubyte), `us` (GLushort), `ui` (GLuint). A més a més aquestes abreviacions poden anar acompanyades al davant d'un nombre indicant el nombre de paràmetres d'aquell tipus, per exemple: `glColor4f()` és la funció per definir el color amb 4 GLfloat; o seguides d'una `v` per indicar que el paràmetre és un vector, per exemple: `glLightfv()`.

D.6 La Vista

Una cosa que passa molt sovint desapercebuda al començar a treballar amb OpenGL és com es representa del contingut sobre la vista. L'OpenGL, per defecte, representa el contingut amb el punt $(0, 0, 0)$ al centre de la pantalla i una unitat d'amplada a cada banda, sigui quina sigui la proporció de la vista. Mirant sempre en direcció cap avall $(0, 0, -1)$.

És el programador, qui ha d'encarregar-se de definir el tipus de vista (ortogràfica o perspectiva), les proporcions respecte la pantalla, així com girar i desplaçar el punt de vista.

D.6.1 Vista Ortogràfica

El següent és un exemple de com configurar una vista ortogràfica. A partir de l'amplada i l'alçada de la vista es calcula la relació d'aspecte i les dimensions de l'amplada. També s'hi aplica una rotació de 45° respecte diferents eixos per observar com canvia.

```
// Establim la projecció ortogràfica
CGRect rect = [self bounds];
GLfloat rel = rect.size.width / rect.size.height;
glMatrixMode(GL_PROJECTION);
glOrthof(-2, 2, -2/rel, 2/rel, -10.0, 10000.0);
glRotatef(0, 0, 0, 0);
glViewport(0, 0, rect.size.width, rect.size.height);
// Dibuixem els eixos
```

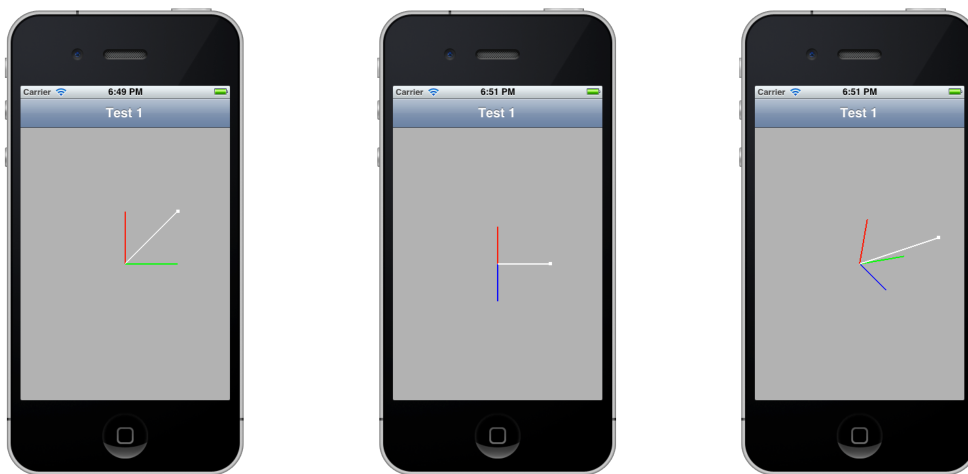

Figura D.3: Ortogràfica: glRotate(0,0,0,0) — (45,1,0,0) — (45,1,1,0)

És important entendre que cada transformació que es fa s'acumula a l'anterior, ja que el sistema fa productes de transformacions (matrius). I també s'ha de veure que l'ordre en que s'apliquen és el contrari al què s'ha declarat. En aquest cas, la projecció queda girada 45°, i l'amplada de la vista va des de -2 fins a 2, i l'alçada proporcionalment a l'amplada.

D.6.2 Vista Perspectiva

Aquest altre exemple mostra com configurar una vista en perspectiva. Com s'ha dit anteriorment, es considera que la càmera, l'ull o el punt de vista —com se'n vulgui dir— és al punt $(0, 0, 0)$ i mira cap avall $(0, 0, -1)$. A partir d'aquí s'ha de treballar amb l'amplada i l'alçada de la vista que s'ha definit per trobar-ne la relació d'aspecte, i només queda per definir en quin punt es projecta la imatge per definir-ne'n les dimensions. El punt on es projecta (`near`) és la distància des de la càmera en direcció a on es mira, i pot ser qualsevol major de zero. Un cop establer es senzill calcular l'amplada de la vista de projecció a partir de l'angle d'enfocament de la càmera —és un simple producte de la tangent de l'angle. En l'exemple també s'hi han aplicat diverses rotacions per veure'n les diferències.

```
// Establim la projecció ortogràfica
CGRect rect = [self bounds];
GLfloat near = 3;
GLfloat rel = rect.size.width / rect.size.height;
GLfloat size = near * tanf(RADIANS(45.0) / 2.0);
glMatrixMode(GL_PROJECTION);
```

```
glFrustumf(-size, size, -size/rel, size/rel, near, 1000.0);
glTranslatef(0.0, 0.0, -4.0);
glRotatef(0, 0, 0, 0);
glViewport(0, 0, rect.size.width, rect.size.height);
// Dibuixem els eixos
```

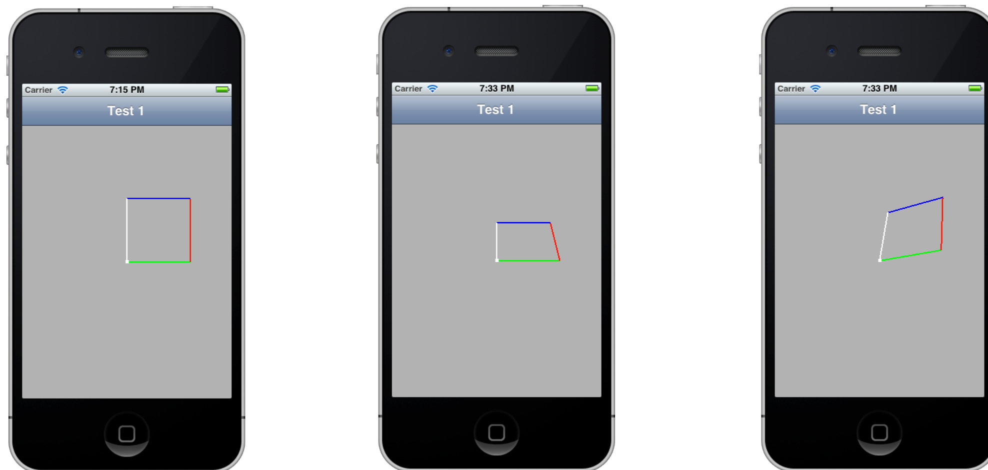

Figura D.4: Perspectiva: glRotate(0,0,0,0) — (-45,1,0,0) — (-45,1,1,0)

En aquest cas també s'ha traslladat el punt de vista amunt, perquè el quadrat estava dibuixat en $z = 0$ i no s'hagués vist.

D.7 Dibuixat

El dibuixat tot hi ser el nucli fort de funcionament de l'OpenGL, també n'és el més senzill d'entendre, simplement coneixent-ne els elements que s'hi poden dibuixar: punts, línies i triangles; i que tots estan representats per vèrtexs definits amb coordenades cartesianes.

Com s'ha dit, l'OpenGL ES és una versió reduïda i optimitzada de l'OpenGL i per tant només s'han implementat les funcions més genèriques i alhora optimitzades. A continuació se'n descriuen un parell que serveixen per a la majoria de casos.

D.7.1 Dibuixar Arrays

La primera funció és `glDrawArrays`, la qual permet dibuixar diferents objectes de diverses maneres, a partir d'un array de vèrtexs. Aquesta n'és la seva declaració:

```
glDrawArrays (GLenum mode, GLint first, GLsizei count);
```

El `mode` indica què i com es dibuixa a partir del llistat de vèrtexs que s'han d'haver introduït amb anterioritat. Els modes disponibles són: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` i `GL_POLYGON`; els quals permeten dibuixar punts, línies (soles, contínues i/o tancades), triangles (sols, continus i/o en ventall), quadrats (sols o continus) i polígons (només sols). El fet que siguin continus vol dir que aprofiten punts anteriors per generar nous triangles, línies i quadrats.

El paràmetre `first` indica el primer vèrtex a tenir en compte, i el paràmetre `count` el nombre de vèrtexs que s'utilitzaran. Cada mode de funcionament va agafant els vèrtexs que li calen fins que s'acaben, per tant és tasca del programador ficar-los en l'ordre correcte.

La seva utilització es pot veure en aquest petit tros de codi, on s'hi observa la definició dels vèrtexs com un array de `GLfloat`, l'activació de la funcionalitat que permet enviar un array de vèrtexs en un buffer de la targeta gràfica mitjançant la funció `glVertexPointer`. També s'hi han afegit instruccions per indicar el color i la dimensió del punt

```
GLfloat punts[12] = {
    0.0, 0.0, 0.0,
    0.5, 0.0, 0.0,
    0.0, 0.5, 0.0,
    0.5, 0.5, 0.5
};

// Dibuixem el triangle vermell
glEnableClientState(GL_VERTEX_ARRAY);
glColor4f(1.0, 0.0, 0.0, 1.0);
glVertexPointer(3, GL_FLOAT, 0, triangle);
glDrawArrays(GL_TRIANGLES, 0, 3);

// Dibuixem el punt verd
glColor4f(0.0, 1.0, 0.0, 1.0);
glPointSize(3.0);
glDrawArrays(GL_POINTS, 3, 1);
glDisableClientState(GL_VERTEX_ARRAY);
```

D.7.2 Dibuixar Elements

Semblant a la funció anterior, hi ha la funció `glDrawElements` que permet dibuixar elements. Mentre l'anterior anava utilitzant els vèrtexs en ordre, en aquest cas se li indica amb un llistat d'índexs quin vèrtex utilitzar en cada moment. Això és molt útil per a re-utilitzar vèrtexs en diferents cares, com per exemple en un cub, que té 8 vèrtexs i cada vèrtex forma part de 3 cares diferents. La seva declaració és aquesta:

```
glDrawElements (GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);
```

El paràmetre `mode` funciona igual que en la funció anterior, el paràmetre `count`, semblant a l'anterior, indica el nombre de vèrtex a utilitzar però comptats en els índexs. A banda, hi ha el paràmetre `type` que indica el tipus de variable dels índexs: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT` o `GL_UNSIGNED_INT`. Es recomana sempre utilitzar el més petit possible. Per acabar el paràmetre `indices` és un array a els índexs del tipus especificat.

D.8 La Llum

Aquesta secció introduieix el funcionament de la llum dins de l'OpenGL. Les següents imatges mostren un cub amb la il·luminació desactivada, amb la il·luminació activada però sense cap llum, activada sense material, i activada amb llum i material definit.

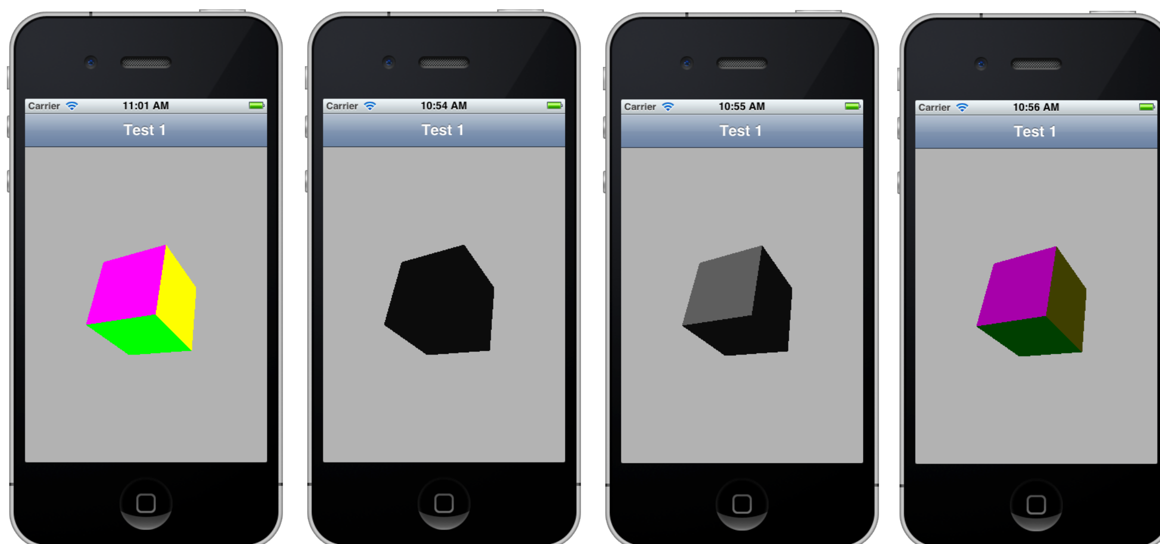

Figura D.5: Il·luminació: Desactivada — Sense Llum — Sense Material — Amb Tot

D.8.1 Activar la Il·luminació

Encara que sembli estrany, per defecte l'OpenGL no treballa amb llums, això fa que els objectes que s'hi dibuixen es mostrin en el seu color original, que es podria considerar com si hi hagués una llum d'ambient del 100%. La il·luminació s'activa amb la comanda `glEnable(GL_LIGHTING);`, però quan es fa això tots els objectes excepte el fons queden negres. Això és degut a que s'ha activat la il·luminació, però no s'hi ha afegit cap llum.

També s'han d'activar cadascun dels llums amb la comanda respectiva `glEnable(GL_LIGHT0)`, que activa el llum respectiu (del 0 al 7). Un cop s'ha activat un llum també s'ha de configurar o definir.

D.8.2 Components de color

Uns dels paràmetres més importants a l'hora de configurar un llum és indicar-ne el color de la llum. Hi ha tres components de color que es poden configurar en un llum:

- El **difús** és el color de la llum pròpiament dita, la que toca sobre la superfície dels objectes.
- L'**especular** és el color de la llum en els reflexos produïts en les cares perpendiculars respecte la direcció de la llum.
- L'**ambient** és el color de la llum que toca les superfícies de forma indirecta.

El següent és un exemple senzill de com es configuren els tres components d'un llum.

```
const GLfloat light0Diffuse[] = {0.05, 0.05, 0.05, 1.0};
const GLfloat light0Specular[] = {0.05, 0.05, 0.05, 1.0};
const GLfloat light0Ambient[] = {0.05, 0.05, 0.05, 1.0};
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0Diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0Specular);
glLightfv(GL_LIGHT0, GL_AMBIENT, light0Ambient);
```

D.8.3 Posició

Un altre dels paràmetres importants és la posició del llum, que no necessita explicacions més profundes. Amb un exemple n'hi ha prou:

```
const GLfloat light0Position[] = {10.0, 10.0, 10.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light0Position);
```

D.8.4 Atenuació

L'últim dels paràmetres comuns a considerar és el de l'atenuació, que es pot establir com a amb una del següents tres opcions: `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` i `GL_QUADRATIC_ATTENUATION`. Per entendre'n el funcionament, el millor és provar-los, tot i que no es difícil suposar-ho amb els seus noms.

D.8.5 Llum direccional

Fins ara no s'ha indicat en quina direcció va la llum, ja que per defecte el llum il·lumina en totes direccions. Si es vol crear un focus, s'han de configurar els següents paràmetres:

- `GL_SPOT_DIRECTION` configura en quina direcció s'il·lumina.
- `GL_SPOT_EXPONENT` indica la intensitat de la llum i com s'atenua amb la distància.
- `GL_SPOT_CUTOFF` configura l'angle d'obertura de la llum del focus mesurat des del seu centre.

D.8.6 Les Normals

Es possible que molta gent, després d'il·luminar l'escena, segueixi veient els objectes de color negre. Descartant algun error en la configuració de la il·luminació, és hora d'introduir el concepte de les normals.

Les normals són uns vectors situats en cada vèrtex què indiquen la direcció perpendicular de la superfície. L'OpenGL per a cada punt d'una cara n'interpola la normal a partir de la dels seus vèrtexs. Així, una cara plana, hauria de tenir la mateixa normal en tots els seus vèrtexs. Per contra, una cara corbada les hauria de tenir lleugerament obertes o tancades depenen si la curvatura és convexa o còncava.

En un superfície corbada formada per diverses cares triangulars, seria convenient aplicar com a normal la mitjana de les normals de totes les cares que passen per aquell punt.

Com que calcular això té un cost, el més interessant és tenir-les calculades per endavant o només calcular-les un cop —quan es crea la superfície. Un cop estan definides s'utilitzen considerant que hi ha d'haver una normal per a cada vèrtex, per tant el seu array serà de la mateixa mida. En el següent exemple es veu com s'activen i s'utilitzen quan es volen dibuixar els elements.

```
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, normals);
...
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_SHORT, indices);
...
glDisableClientState(GL_NORMAL_ARRAY);
```

S'ha d'esmentar també, que en el cas de cares planes —com podrien ser les d'un cub— cada vèrtex té una normal diferent per a cada cara del cub, per tant si es volen utilitzar normals diferents s'ha de repetir

cada vèrtex per cada normal diferent que tingui. En aquests casos, pot ser més interessant no utilitzar cap array de normals, i en el seu lloc utilitzar la comanda `glNormal3f` que defineix la normal per als successius elements a dibuixar.

```
glNormal3f(0.0, 0.0, 1.0);
glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_SHORT, indices);
```

D.9 El Material

En OpenGL quan es defineix un color en realitat s'indica com li reflexa la llum en el món real. Si s'indica que li reflexa la llum vermella, sembla vermell si se l'il·lumina amb una llum blanca, però no si se l'il·lumina amb una llum que no té l'espectre del vermell, en aquest cas es veu fosca ja que no reflecteix gens de llum.

Sabent això l'OpenGL no només permet definir un únic color, sinó que se'n pot definir un per a cada tipus de llum (difusa, especular, ambient). A més a més, es pot definir que els materials emetin llum.

No s'hi aprofundeix massa, ja que és qüestió de fer-ne proves per veure'n l'efecte, però un exemple amb algunes instruccions sempre és benvingut.

```
GLfloat color[] = {0.0, 0.1, 0.9, 1.0};
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, color);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 35.0);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, color);
```

D.10 Textures

S'ha vist en la secció anterior com es treballa amb color, però si el que es vol és un resultat —diguem-ne més professional— és més adequat treballar amb textures què es pinten directament sobre les superfícies, amb la conseqüent millora visual i del rendiment si s'hagués de simular a base de colors. La següent figura mostra un cub que té una cara amb textura i la resta amb colors.

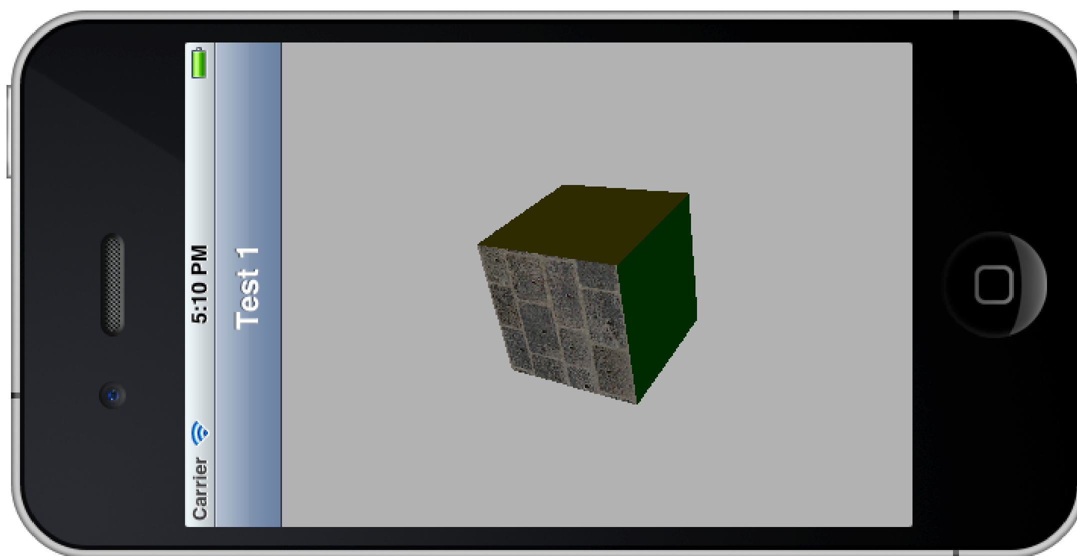

Figura D.6: Il·luminació, Material i Textura

D.10.1 Activar les textures

Per poder utilitzar les textures s'han d'activar les següents característiques a l'OpenGL.

```
glEnable(GL_TEXTURE_2D);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_SRC_COLOR);
```

La primera permet emmagatzemar imatges en dues dimensions (2D) que han de servir per a utilitzar-les com a textura. La segona permet combinar imatges per aconseguir efectes interessants. L'última permet indicar amb el primer paràmetre com s'agafa la informació de la textura, i amb el segon com s'aplica sobre la superfície. Això permet combinar la textura amb el color de la superfície, aplicar-hi certa transparència, entre altres opcions.

D.10.2 Creant textures

Per utilitzar textures, primer s'han de crear i deixar a punt per a que l'OpenGL les pugui utilitzar. Això es pot veure en el següent fragment de codi.

```
GLuint      texture[1];             // array de textures
glGenTextures(1, &texture[0]);
glBindTexture(GL_TEXTURE_2D, texture[0]);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height,
             0, GL_RGBA, GL_UNSIGNED_BYTE, imageData);
```

Les tres primeres instruccions declaren un array d'una textura, la generen i l'enllacen a l'OpenGL. Les altres dues instruccions permet configurar dos paràmetres que per defecte treballen amb MipMaps —que no s'explicaran—, els quals indiquen que el càlcul del color al dibuixar es fa per interpolació lineal de la textura, també es pot configurar per a que s'utilitzi el píxel més proper (`GL_NEAREST`). L'última instrucció envia les dades de la imatge indicant-ne les dimensions i el format.

D.10.3 Limitacions

S'ha de dir que hi ha certes limitacions en la utilització de textures, les quals s'enumeren i es resumeixen a continuació:

- **Quantitat**: El nombre de textures que poden utilitzar-se alhora depèn de la versió i la implementació. Es pot consultar a les característiques.
- **Mides**: L'OpenGL necessita treballar amb imatges de dimensions potència de 2, és a dir (2, 4, 8, 16, 32, 64, 128, 256, 512 o 1024), per exemple: 64x128 ó 256x256.
- **Mida**: La mida màxima de cada textura també queda limitada en cada implementació de l'OpenGL. Valor que també es pot consultar.
- **Origen**: El píxel $(0, 0)$ és a la banda inferior esquerra, per tant si es carrega la textura des d'un fitxer d'imatge s'ha d'invertir verticalment abans d'enviar-ne les dades a l'OpenGL.
- **Coordenades**: L'OpenGL no treballa amb píxels, sinó amb coordenades de $(0.0, 0.0) - (1.0, 1.0)$ essent els punts inferior esquerra i superior dret respectivament.

D.10.4 Coodenades

Per aplicar una textura sobre una superfície, s'ha de definir per a cada vèrtex quin punt de la textura hi ha de pintar. A partir d'aquí l'OpenGL interpola la textura per representar la imatge en aquella superfície. Per fer això, com s'ha vist en altres seccions, es pot utilitzar l'opció `GL_TEXTURE_COORD_ARRAY` per a que l'OpenGL ho faci transparentment treballant amb arrays, o amb la instrucció `glMultiTexCoord2f` per a definir-lo per a cada punt.

D.10.5 Altres paràmetres

És pot treballar amb textures activant altres paràmetres què permeten més flexibilitat, com per exemple `GL_TEXTURE_WRAP_X`, què permet aquestes dues opcions:

- `GL_REPEAT`, què fa repeticions d'una textura en els seus diferents eixos.
- `GL_CLAMP_TO_EDGE`, què fa que l'última línia de la textura s'estiri fins al final.

D.11 Transformacions

En l'apartat de la vista ja s'ha vist alguna transformació, ja que les funcions `glOrtho` i `glFrustumf` transformen la matriu actual. En aquesta secció s'hi aprofundeix mostrant la resta d'aquestes funcions de transformació.

S'ha de tenir present en quina matriu s'està treballant en tot moment: la `GL_PROJECTION` és la matriu què defineixen la vista, `GL_MODELVIEW` és la matriu per manipular els objectes, `GL_TEXTURE` és la matriu de les textures i `GL_COLOR` la del color —per descomptat.

S'ha de fer esment, també, que la de `GL_PROJECTION` es multiplica al final per la `GL_MODELVIEW` per aconseguir la vista final. Cal tenir-ho en compte si es treballa amb OpenGL 2.0 i els shaders, ja que és allí on es fa el producte d'aquestes matrius.

D.11.1 Carregar i desar matrius

Les primeres funcions de transformació a considerar són les que carreguen i desen matrius per a utilitzar-les en el moment oportú.

- **glLoadIdentity**, carrega la matriu identitat, és a dir la que no genera cap transformació.
- **glLoadMatrix**, carrega una matriu (4x4) amb els 16 valors passats en un punter com a paràmetre.
- **glPushMatrix**, desa a la pila la matriu que utilitzem, permet seguir fent canvis i després tornar a la versió desada.
- **glPopMatrix**, recupera l'última matriu desada a la pila.

D.11.2 Multiplicar Matrius

La resta de funcions simplement apliquen una nova transformació a l'actual, és a dir es multipliquen amb l'actual. Són aquestes:

- **glFrustum**, transforma el contingut per mostrar-se en perspectiva. Cal cridar-la al principi, per a que sigui l'última en aplicar-se.

- **glOrtho**, transforma el contingut per mostrar-se en una vista ortogràfica. També S'ha de cridar al principi.
- **glMultMatrix**, transforma la matriu multiplicant-la amb una nova matriu a partir dels 16 valors passats en un punter com a paràmetre.
- **glRotate**, gira el contingut certs graus al voltant d'un eix que passa pel centre.
- **glScale**, escala el contingut proporcionalment en cada eix, respecte el centre.
- **glTranslate**, desplaça el contingut unes unitats respecte els eixos de coordenades.

D.11.3 Ordre de les Transformacions

És molt important tenir clar l'ordre de les transformacions. Les transformacions no s'apliquen en l'ordre que es programen, sinó en ordre invers. Així el codi següent primer gira el contingut a dibuixar i posteriorment el trasllada.

```
glLoadIdentity();
glTranslate();
glRotate();
```

Si es vol girar un objecte respecte el seu centre, primer s'ha de traslladar el seu centre al punt $(0, 0, 0)$, fer el gir, i tornar-lo a la seva posició anterior; sempre en ordre invers. Per exemple si el centre de l'objecte és al punt $(1, 2, 3)$ i es vol girar 30º respecte l'eix X s'hauria de fer així:

```
glTranslate(1,2,3);
glRotate(30,1,0,0);
glTranslate(-1,-2,-3);
```

On veiem que primer mou l'objecte al centre (última instrucció), el gira 30º respecte l'eix X (segona instrucció) i el torna a la posició original (primera instrucció).

Índex de figures

2.1 Calendari del Projecte 10

3.1 Cubs de Rubik clàssics: 3x3 — 2x2 11
3.2 Esfera de Rubik — Cub deformat de Rubik 12
3.3 Cub de 3x3 amb talls 12
3.4 Figures amb un únic pla: 1 tall — 2 talls — 3 talls 13
3.5 Dos plans: Perpendicular — 60º original — 60º gir vertical — 60º gir inclinat 14
3.6 Talls no perpendiculars: Vista general — Vista zenital — Terra barrejada 14
3.7 Talls perpendiculars: 2 plans — 3 plans 15
3.8 Talls no perpendicular: 3 plans — 4 plans — 5 plans 15
3.9 Sòlids Platònics: Tetràedre — Hexàedre — Octàedre — Dodecàedre — Icosàedre 15
3.10 Tetraedre de Rubik: Original — Barrejat — de 3x3x3 — Plans de tall 16
3.11 Icosaedre tallat per mig 16
3.12 Gir de 180º en nivells de plans perpendiculars 17
3.13 Girs de 60º o de 120º 18
3.14 Posició original — Gir nivell 1 — Gir nivell 0 19
3.15 Esfera blanca — Esfera amb colors — Esfera amb textura 19

4.1 Pla $\alpha : Ax + By + Cz + D = 0$ 25
4.2 Pla $\alpha : z = 0$ i punts: p_0, p_1 i p_2 25
4.3 Plans α i β: Interseccions pla–pla i recta–pla 26
4.4 Posició de la càmera i rotacions en l'escena 27
4.5 Eix de rotació de la càmera 28
4.6 Textura sense Material — Textura amb Material Vermell 30

5.1 Cassos d'Ús — Jugador 32

5.2 Cas d'Ús — Mòdul de Vista 32
5.3 Cas d'Ús — Mòdul de Moviments 33
5.4 Classes — Model-Vista-Controlador 34
5.5 Classes Model — El "Cub de Rubik" 35
5.6 Pantalla inicial: Llistat — Edició — Esborrat 37
5.7 Pantalla de Joc 37

6.1 RBK9ListController: Llistat — Edició — Eliminar 39

A.1 Captura de l'arrancada del programa 66
A.2 Captura jugant amb el llistat 67
A.3 Captura al seleccionar un "Cub" 67
A.4 Captura al girar la càmera 68
A.5 Captura al moure la càmera 68
A.6 Captura del gest de moure peces 69
A.7 Captures de l'animació del moviment 69

B.1 Cassos d'Ús — Modelador 71
B.2 Cassos d'Ús — Dissenyador 71
B.3 Cas d'Ús — Mòdul de Creació 72
B.4 Cas d'Ús — Mòdul de Modelat 73
B.5 Cas d'Ús — Mòdul de Pintat 73
B.6 Cas d'Ús — Mòdul de Tallat 74
B.7 Cas d'Ús — Mòdul de Peces 74
B.8 Pantalla d'Edició — Dibuixat — Canvi de Mode 75
B.9 Pantalla d'Edició — Mode de Dibuixat 76
B.10 Pantalla d'Edició — Mode de Modelat 76
B.11 Pantalla d'Edició — Mode de Pintat 77
B.12 Pantalla d'Edició — Mode de Tallat 77
B.13 Pantalla d'Edició — Mode de Test 77

D.1 OpenGL Embedded Sistems sobre iOS 84
D.2 Depth Buffer: Desactivat — Activat 86
D.3 Ortogràfica: glRotate(0,0,0,0) — (45,1,0,0) — (45,1,1,0) 88

D.4 Perspectiva: glRotate(0,0,0,0) — (-45,1,0,0) — (-45,1,1,0) 89

D.5 Il·luminació: Desactivada — Sense Llum — Sense Material — Amb Tot 91

D.6 Il·luminació, Material i Textura 93