Universitat de Girona
Escola Politècnica Superior

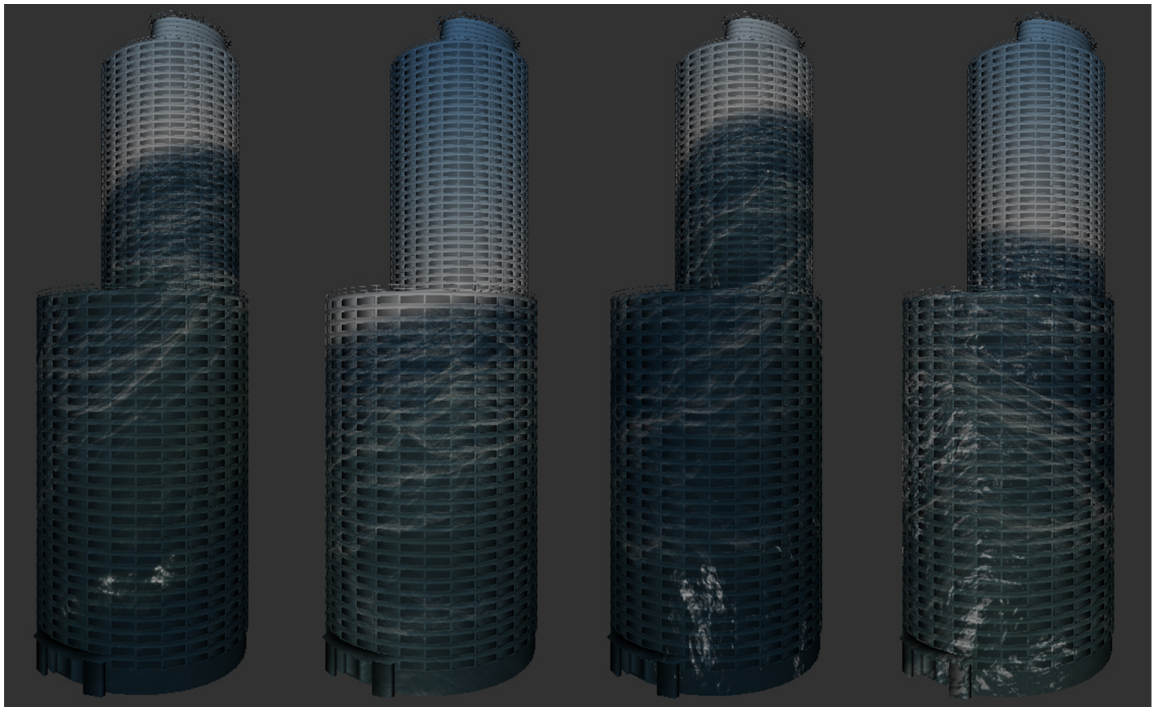
Escola Politècnica Superior

Universitat de Girona

Projection Mapping amb Shadertoy i openFrameworks

Treball final de grau

Grau en Enginyeria Informàtica



Document: Memòria

Alumne: Lluís Borràs Massaguer

Tutor: Gustavo Patow i Gonzalo Besuievsky

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: LSI

Convocatòria (mes/any): Juny 2015

Índex

Capítol 1

Introducció

En aquest primer capítol explicarem els inicis que van dur a decidir desenvolupar aquest projecte, quines motivacions o necessitats teníem, quin era el propòsit d'elaborar el projecte i finalment quins objectius es volien assolir en aquest treball final de grau. També presentarem com s'han organitzat els diferents apartats d'aquest treball.

Abans de començar, però, posarem en context aquest treball. El *projection mapping* és una tècnica que tracta de projectar imatges o vídeo, en 2D o 3D, sobre superfícies irregulars, ja siguin edificis o qualsevol tipus de mobiliari (Figura 1.1). Es pot interactuar amb l'objecte, per tal d'encaixar les imatges amb la superfície, gràcies a programari dedicat exclusivament a aquest tipus de projeccions: 'Resolume Arena', 'Mad Mapper', 'arkaos GrandVJ XT' ...

Figura 1.1: *Projection mapping* a la Sagrada Familia a càrrec de 'Moment Factory'

1.1 Motivacions

Per norma general, normalment en un treball, recerca, estudi, etc., hi ha dos tipus de motivacions amagades al darrera: les personals i les necessitats professionals. Les personals solen estar més encarades en les ganes d'hom que té de treballar o investigar en un tema en concret. En aquest cas aquesta motivació neix, segurament, de l'admiració: últimament, en moltes presentacions, festes culturals, esdeveniment de màrqueting, etc., els organitzadors utilitzen un *projection mapping*, ja sigui perquè és bonic de veure o perquè és un bon reclam publicitari. En els mesos passats n'hem pogut veure a la Catedral de Girona, a la Sagrada Família de Barcelona, en diferents presentacions publicitàries d'arreu del món, etc. Els *projection mappings* aconsegueixen atreure a molta gent i, en el meu cas, volia saber quin tipus de feina s'hi amagava darrera.

El *projection mapping* és un món molt complex, cada projecció explica una història diferent i hi ha molts perfils de persona treballant-hi al darrera, des del dissenyador gràfic, passant pels tècnics de vídeo i so fins a l'autor de la història. Les aplicacions que s'utilitzen per fer *projection mapping*, de les quals les més conegudes: 'Resolume Arena', 'Mad Mapper' i 'arkaos GrandVJ XT' són aplicacions comercials a gran escala. És un tipus de *software* molt específic i, per tant, també repercuteix en el preu de la llicència.

Els vídeo DJ's (VDJ a partir d'ara) que volen treballar amb programari lliure es troben que ho fan en un entorn bastant caòtic, ja que no existeix software lliure equiparable al programari mencionat anteriorment, i per fer-ho es troben immersos en utilitzar diferents programes lliures que acaba costant més sincronitzar-los tots entre si, que no pas aconseguir resultats, a més que no tots treballen de la mateixa manera utilitzant la mateixa *pipeline*, i sempre hi ha problemes de compatibilitat.

1.2 Propòsit

Vista la problemàtica que tenen els VDJ que volen utilitzar programari lliure per a les projeccions, el propòsit principal d'aquest treball és el de desenvolupar una aplicació, tal que garanteixi una *pipeline* de treball senzilla i eficaç. L'aplicació permetrà tenir una previsualització de com podria ser la projecció final utilitzant un projector en models a gran escala. Per això l'aplicació permetrà carregar models 3D per poder-hi 'mapejar' imatges. Es requerirà d'un ordinador per càlcul suficientment potent.

1.3 Objectius

L'objectiu principal d'aquest treball és desenvolupar una aplicació on es puguin previsualitzar projeccions de *projection mapping*, mitjançant la tecnologia dels *shaders* (GLSL) sobre models 3D.

A més, com a objectius secundaris l'aplicació permetrà als VDJ que vulguin fer les projeccions amb programari lliure tinguin una *pipeline* de treball robusta. Per a fer-ho, principalment treballarem amb *shaders* (Secció 5.3), del projecte 'Shadertoy' (Secció 7.1). Aquests *shaders* actuaran de projecció, com el vídeo que es projecta en un *projection mapping* real. Per fer-ho viable, s'haurà d'estudiar el motor del 'Shadertoy', com construeix els *shaders* i com els executa des de la plataforma on-line.

Tots els *shaders* de 'Shadertoy' estan identificats amb un ID únic. Aquest ID el podem trobar en el mateix enllaç, és a dir https://www.shadertoy.com/ID. Això ens dóna la possibilitat d'estudiar una manera per poder descarregar de manera senzilla, a partir de l'ID, el *shader*. Així doncs, un altre objectiu serà estudiar com poder descarregar un *shader* al disc dur. Per a fer-ho s'estudiarà el llenguatge de programació Python (Secció 7.5).

Amb aquests conceptes, el més important és trobar un entorn de treball adequat i, en aquest cas, s'estudiaran diferents *frameworks*, com poden ser 'Freeframe' o 'openFarmeworks' especialment enfocats en codificació d'eines de vídeo i so.

Per tant, les tasques principals d'aquest treball són:

- Estudiar una metodologia o *pipeline* de treball per a simular *projection mapping*.
- Estudiar els conceptes bàsics d'informàtica gràfica i GPU.
- Estudiar els conceptes més importants d'*OpenGL*.
- Estudiar un nou llenguatge: el Python.
- Implementació d'una eina que permeti obtenir *shaders* de 'Shadertoy'.
- Estudi i recerca de la funcionalitat d'un *framework* i les seves llibreries o *addons* incorporats adequats per al projecte.
- Implementació d'una eina que permeti la simulació d'un *projection mapping*.
- Estudiar la millor manera per a desenvolupar una interífice gràfica per aquest treball.
- Elaborar la documentació de tots els capítols del projecte.

1.4 Capítols de la memòria

La memòria està estructurada en els següents apartats:

- **Capítol 1. Introducció**. S'expliquen les motivacions, els propòsits i els objectius del projecte. També es presenta un resum dels diferents apartats del projecte.

- **Capítol 2. Estudi de viabilitat**. Es justifica la viabilitat del projecte en terme de recursos tant econòmics, tecnològics i humans.

- **Capítol 3. Metodologia**. S'explica la metodologia seguida per a desenvolupar el projecte.

- **Capítol 4. Planificació**. S'explica el *timing* del projecte, és a dir, el pla de treball, les tasques planificades i el temps estimat per cada etapa.

- **Capítol 5. Marc de treball i conceptes previs**. S'introdueixen els conceptes teòrics necessaris per a comprendre millor la resta de la memòria.

- **Capítol 6. Requisits del sistema**. Es descriuen els requisits funcionals i no funcionals del sistema per assolir els objectius.

- **Capítol 7. Estudis i decisions**. Es descriu el programari, el *framework*, l'IDE i les llibreries utilitzades en aquest projecte.

- **Capítol 8. Anàlisi i disseny del sistema**. S'expliquen les funcionalitats que haurà d'adquirir el sistema a partir de les necessitats dels usuaris, així com les solucions a partir dels diagrames dissenyats.

- **Capítol 9. Implementació i proves**. S'explica com s'ha implementat l'aplicació, amb els detalls més importants, i com s'han anat solucionat els problemes sorgits a partir d'exemples i proves gràfiques.

- **Capítol 10. Resultats**. Es mostren els resultats obtinguts, a partir d'exemples i imatges de l'aplicació.

- **Capítol 11. Conclusions**. S'exposen les conclusions de l'assoliment dels objectius i de l'assoliment dels requisits del projecte, així com una crítica dels resultats obtinguts.

- **Capítol 12. Treball futur**. S'expliquen les possibles ampliacions, millores o treballs futurs que es poden realitzar a partir de l'aplicació obtinguda.

- **Bibliografia**. Llistat de les referències utilitzades per desenvolupar el projecte.

- **Capítol 13. Manual d'usuari i instal · lació**. S'explica com s'ha d'utilitzar i instal · lar l'aplicació en un ordinador.

Capítol 2

Estudi de viabilitat

En aquest capítol es valorarà la viabilitat del projecte des de diferents aspectes com poden ser: els recursos tècnics, els recursos humans, els requisits tecnològics o el cost econòmic.

2.1 Recursos tècnics

Aquest treball bàsicament s'ha realitzat sempre sobre la tecnologia d'un mateix ordinador, tenint en compte que el sistema operatiu d'aquest és un Windows 7 de 64 bits i no és compatible ni amb Linux ni amb Mac OS.

Pel que fa als components de l'ordinador, és una màquina amb un processador AMD FX-8350 de 8 nuclis a 4GHz en un sistema operatiu Windows 7 de 64 bits, com ja s'ha esmentat. Pel que fa a la targeta gràfica és un model AMD Radeon R7 260X de 2GB GDDR5 (assemblada per *Sapphire*) amb tecnologia OpenCL. Disposa de dues memòries RAM DDR3 de 4GB a 1866Hz cadascuna.

2.2 Recursos humans

Com la majoria de projectes informàtics, aquest treball requereix de tres perfils de persona:

- **Analista**: Serà l'encarregat de definir com haurà de ser l'aplicació, quines funcionalitat haurà de tenir i com s'haurà de dur a terme per aconseguir-les. També tindrà la responsabilitat d'estudiar les eines que es necessitaran juntament amb la seva documentació. Finalment s'encarregarà d'elaborar la memòria del projecte.

- **Programador**: Serà l'encarregat de fer possible la creació de l'aplicació. A partir de les consignes de l'analista haurà de ser capaç d'entendre i desenvolupar l'aplicació a partir de les eines aportades.

- **Cap del projecte**: Serà l'encarregat de coordinar el projecte seguint la metodologia de treball escollida (Capítol 3), ajustar els terminis de les diferents etapes de desenvolupament i aconseguir un bon funcionament en el treball.

Com que aquest projecte ha estat realitzat per una mateixa persona li han correspost els perfils d'analista i programador. Pel que fa al perfil de cap de projecte l'han dut a terme els tutors del projecte.

2.3 Requisits tecnològics

A part de la màquina especificada a l'apartat de 'Recursos tècnics', faran falta alguns altres requisits tecnològics per dur a terme aquest treball. S'han utilitzat dos llenguatges de programació per a dur-lo a terme: Python (Secció 7.5) i C++ (Secció 7.4). En el cas del Python necessitarem tenir el compilador instal · lat a la màquina i pel C++ un IDE per poder programar i compilar, en aquest cas s'ha utilitzat *Code::Blocks* (Secció 7.2) de codi obert. També ens haurem d'assegurar de tenir OpenGL amb el *drivers* de la targeta gràfica actualitzats.

Més endavant, a la Secció 7.3, veurem quin *framework* necessitarem per treballar i amb quines llibreries i *addons*.

2.4 Cost econòmic

En aquesta secció definirem el cost econòmic dels recursos tècnics, dels recursos humans i de la tecnologia que hem explicat en els apartats anteriors.

En el cas dels recursos tècnics, com que hem treballat amb un ordinador personal ens estalviarem el cost de comprar-lo, tot i això haurem de tenir en compte l'amortització dels recursos utilitzats. Per calcular-la utilitzarem la següent fórmula:

$$Amortització = \frac{preu\ recurs \cdot mesos\ feina}{36} = \frac{648,90 \cdot 8}{36} = 144,2€$$

Pel que fa al *software*, un dels objectius d'aquest treball era treballar amb programari de codi obert, per tant no suposarà cap cost econòmic afegit.

En el cas dels recursos humans hem vist que, a banda del cap del projecte, són necessaris dos perfils de persona: l'analista i el programador. En aquest cas, tots dos perfils han estat duts a terme per la mateixa persona, però si haguéssim de calcular el cost per independent tindríem que:

- Sou de l'analista: 20€/h
- Sou del programador: 10€/h

Feines a realitzar:

- Estudi de les eines.
- Disseny de la *pipeline* de l'aplicació.
- Disseny de l'aplicació en funció del *framework*.
- Implementació.
- Proves.
- Documentació.

Per tant, el cost en funció de les tasques a realitzar és el següent:

Tasques	**Perfil**	**Hores**	**Cost/hora**	**Cost total**
Estudi de les eines	Analista	200	20	4000
Disseny de la *pipeline* de l'aplicació	Analista	20	20	400
Disseny de l'aplicació en funció del *framework*	Analista	100	20	2000
Implementació	Programador	200	10	2000
Proves	Programador	100	10	1000
Documentació	Analista	100	20	2000
Total Programador		300	10	3000
Total Analista		420	20	8400
Total amortització PC				144
TOTAL				11544

2.5 Conclusió de la viabilitat del projecte

Com que el treball es durà a terme amb un equip que ja prèviament teníem, no suposarà haver de fer un despesa extra. Pel que fa al programari necessari l'objectiu és utilitzar software *open source* i tampoc suposaria una despesa a tenir en compte.

Pel que fa a la remuneració dels treballadors, en el cas hipotètic d'una contractació real, tampoc seria del tot ajustada envers el que podem veure a la taula anterior, ja que aquest treball moltes de les hores han estat invertides en fer recerca, investigar i provar noves tecnologies de les que no se'n tenia un coneixement previ. Amb això, la contractació de treballadors sèniors o professionals en el sector, no hagués suposat un preu tant elevat en nombre d'hores d'anàlisi i segurament també menys en programació.

Com a conclusió, com que és un projecte d'estudiant i es disposa del requisits tecnològics, el treball serà viable.

Capítol 3

Metodologia

Actualment hi ha moltes metodologies de desenvolupament eficients i diferenciables, des de les més antigues com la metodologia *Waterfall* fins a les més modernes tècniques *Agile*. Després d'estudiar diferents metodologies tals com *Spiral*, *Scrum* o *Iterative*, i les esmentades anteriorment, s'ha decidit no fer-ne servir en concret cap d'aquestes.

El que s'ha fet ha estat definir un tipus de metodologia que funcionés bé amb les característiques del projecte, tal com es mostra en el diagrama de flux de la Figura 3.1.

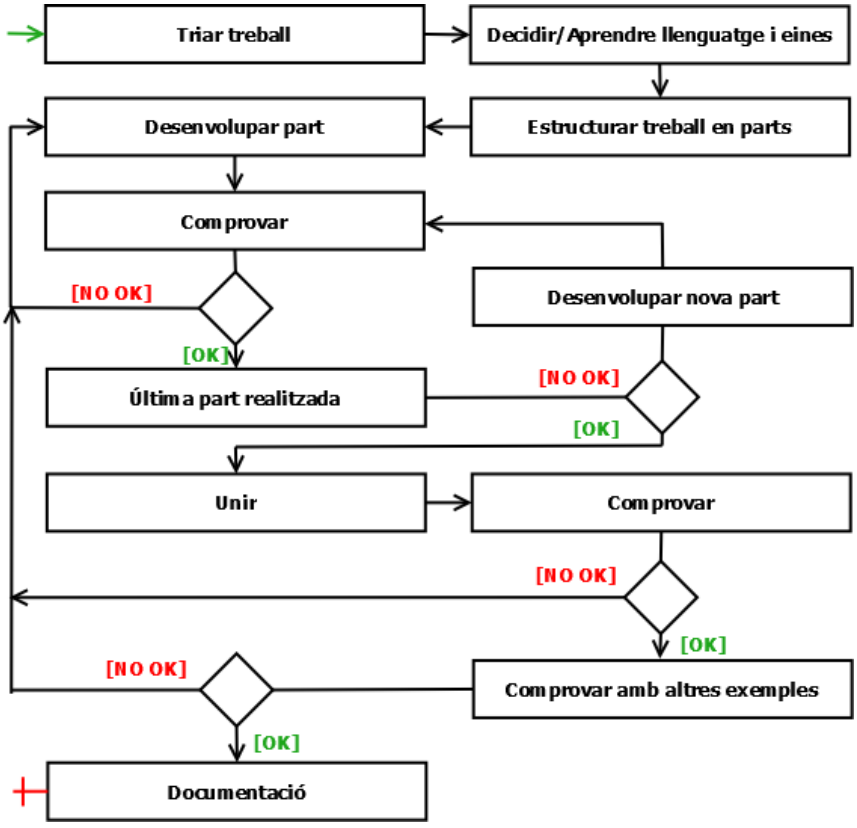


Figura 3.1: Diagrama de flux de la metodologia utilitzada

1. Triar el treball a desenvolupar.
2. Decidir el llenguatge de programació i les eines a utilitzar.
3. Aprendre el llenguatge de programació i el funcionament de les eines escollides.
4. Estructurar el treball en parts segons les funcions que ha de realitzar.
5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.
6. Fer comprovacions per tal de confirmar que el funcionament és correcte al finalitzar la part.
   - 6.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en la última part desenvolupada o en les anteriors, si és convenient.
   - 6.2. Si al fer les comprovacions el resultat és l'esperat, es desenvolupa la part següent tornant al punt 5, si s'han finalitzat les parts amb les seves respectives comprovacions s'inicia el punt 7.
7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.
   - 7.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.
   - 7.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 8.
8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.
   - 8.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.
   - 8.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 9.
9. Acabar la documentació.

Tal com es pot veure consisteix en dividir el projecte en mòduls i organitzar en el temps el seu desenvolupament, segons el temps de verificació i de correcció. Tant durant el temps de desenvolupament com de verificació, es fa un seguiment mitjançant tutories setmanals o bisetmanals depenent de l'etapa, ja que en els inicis són més lents que no pas en les etapes finals, i no sempre es necessari fer tutories setmanalment.

Quan s'acaba un mòdul, sempre que es pugui, es finalitza totalment de manera que no es torna a tocar. D'aquesta manera es garanteix que els errors que puguin sorgir en la resta de mòduls són únicament d'aquests i no de cap dels anteriors. I si es dóna el cas que ho és, al menys s'han minimitzat al màxim els efectes que s'hagin pogut propagar.

Pel procés de disseny s'utilitzarà el llenguatge de modelat estàndard dins el camp de l'enginyeria de programari, l'UML (*Unified Modeling Language*). L'UML s'utilitza per definir un sistema, per detallar els seus elements, per documentar i construir. Per aconseguir això, l'UML disposa de nombrosos tipus de diagrames que mostren diversos aspectes dels elements representats.

Capítol 4

Planificació

4.1 Planificació inicial

La planificació d'aquest treball ha estat influenciada per la metodologia de treball escollida. A la primera trobada, del 2 d'octubre del 2014, van posar-se sobre la taula diverses propostes de projecte que estaven obertes, tot i això la següent setmana es va apostar per entrar en el món del *projection mapping*.

Com que era un camp on no s'hi havia entrat mai, ni per conèixer el funcionament, era un treball que requeria pràcticament començar de zero. Com en tot treball, es va estructurar la primera meitat del desenvolupament en investigar i decidir quins llenguatges i quines eines s'utilitzaven per a *projection mapping*, amb la sort de trobar i poder contactar amb un expert del sector. A partir d'aquest moment, tenint la referència d'una aplicació web com 'Shadertoy', es va decidir desenvolupar una aplicació que permetés encastar els *shaders* de 'Shadertoy' en un *framework* conegut pels VDJs, i que es poguessin "projectar" sobre models 3D.

Així doncs, amb aquestes idees de base es va dissenyar la planificació de treball següent:

1. Estudiar en profunditat el concepte de *projection mapping* i com es realitza, quins programaris hi ha, quines empreses s'hi dediquen, quins conceptes i càlculs s'utilitzen, etc.
2. Contacte amb un expert i coneixedor de les eines del *projection mapping* per encaminar el projecte.
3. Estudiar diferents entorns de treball per a *projcetion mapping* i escollir-ne un.
4. Estudiar els conceptes bàsics de Python per a poder realitzar un petit *script* per descarregar *shaders* de 'Shadertoy'.
5. Implementació de l'aplicació amb les funcionalitats necessàries (*shaders*, models 3D, etc.) i disseny i implementació de la interfície gràfica.
6. Documentar tot el projecte durant el desenvolupament.

A la Figura 4.1 es veuen aquestes tasques repartides en el temps.

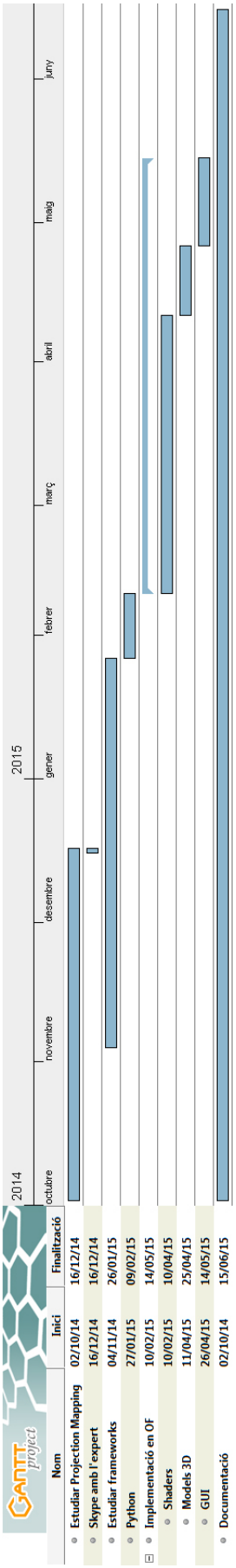


Figura 4.1: Diagrama de Gantt de la planificació inicial

4.2 Planificació final

Tot i la planificació inicial, sovint en un projecte les coses no surten com un ho planifica i voldria que sortissin des del principi. Sí que la planificació en general s'acaba respectant, però apareixen moltes sub-tasques i algun contratemps que s'ha de solucionar. Així doncs, finalitzat el treball la distribució de tasques dins la planificació va quedar com es pot veure a la Figura 4.2.

A partir del 9 d'octube de 2014, quan es va tenir decidit el tema del treball, vam investigar quines empreses treballen en aquest món, i quin tipus de programari solen utilitzar. Ens n'adonem que tots els professionals treballen amb programaris de pagament molt sofisticats. Contactem amb un expert que treballava en el camp en aquell moment i ens explica la seva experiència i propostes de coses a millorar. Després d'aquella trobada, hi havia molts fronts oberts i possibilitats per tractar. Gràcies al coneixement de l'existència d'una aplicació com 'Shadertoy', s'escull treballar amb aquesta eina per desenvolupar el projecte.

La idea principal era poder portar els *shaders* que els dissenyadors penjaven a 'Shadertoy' a l'ordinador local, i a continuació empotrar-los en models 3D per fer la projecció. La idea em va agradar molt, ja que es continuava el temari que s'havia vist a l'assignatura d'Informàtica Gràfica, i a més tenia relació amb el tema escollit, el *projection mapping*.

Decidit que es duria a terme en el treball comencem a investigar quines eines de codi obert utilitzen els VDJs, i ens trobem amb varis *frameworks*: FreeFrame, Cinder, openFrameworks (Secció 7.3) i VVVV. Comencem un període d'investigació de tots ells, perquè hauríem de decidir quin seria el millor per a la nostra aplicació. La idea era utilitzar FreeFrame, ja que era un entorn que l'expert hi havia estat treballant, però vam adonar-nos que estava obsolet. Cinder i VVVV eren molt bons *frameworks* i haguéssin pogut complir les expectatives, però vam acabar escollint openFrameworks per l'ampli contingut de llibreries, la bona documentació i la gran activitat en el fòrum.

openFrameworks té la peculiaritat que ve amb el codi per defecte, però a més s'hi pot adjuntar *addons*, que són llibreries desenvolupades per usuaris a partir del codi que proporciona openFrameworks. Vam trobar que hi havia molts *addons* per dur a terme el treball, i vam decidir que era la millor opció.

Amb això vam començar a implementar l'aplicació per poder executar *shaders* sense textures, a partir dels codis de 'Shadertoy', per ara copiant-los de manera manual. Seguidament vam afegir-hi la possibilitat de passar *uniforms* als *shaders*, uns dels quals les textures, que van portar certa problemàtica que explicarem en capítols següents. Es va començar a estudiar com funcionava internament 'Shadertoy' per l'execució, i així poder estudiar les funcionalitat del Python per poder fer connexions HTTP i poder descarregar els *shaders*.

Amb això complert, es va començar a desenvolupar la interfície gràfica integrada a openFrameworks alhora que s'estudiava el funcionament per poder carregar models 3D a l'aplicació i poder-hi "projectar" els *shaders*. Es va millorar la GUI per tenir la possibilitat de carregar o no els models i es van arreglar alguns *bugs* que van aparèixer en la "projecció" dels *shaders*.

Amb tot acabat i en funcionament, es va passar a documentar tot el treball realitzat.

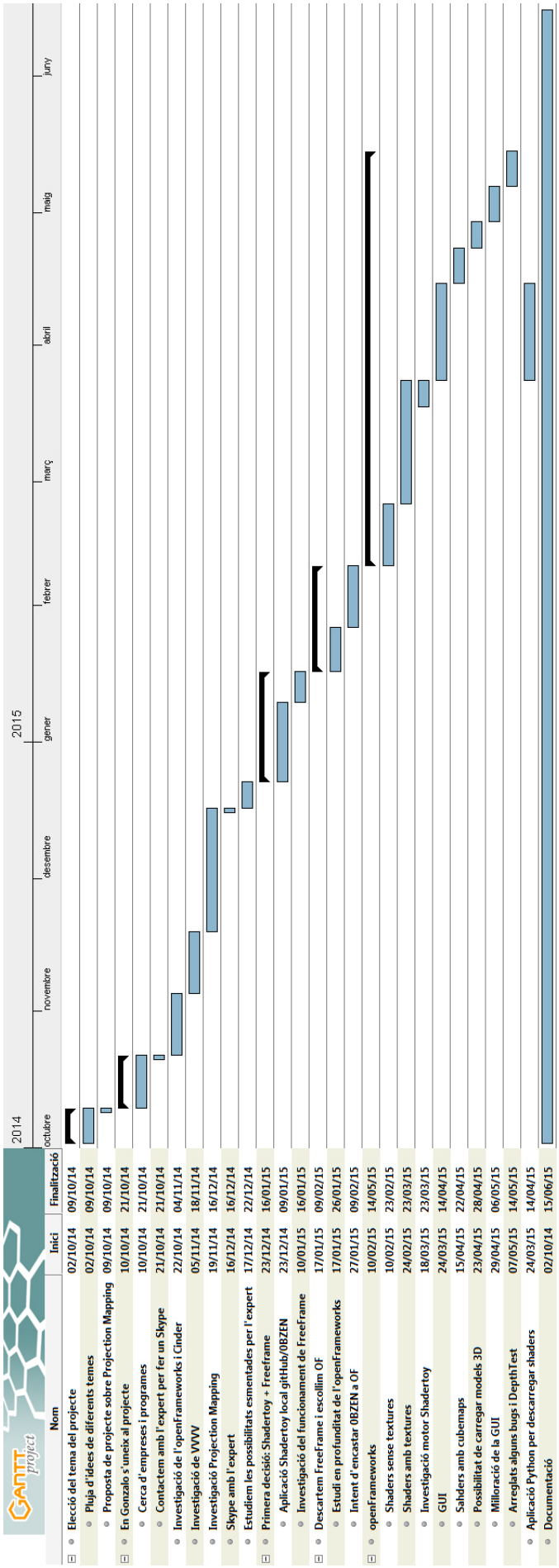


Figura 4.2: Diagrama de Gantt de la planificació final

Capítol 5

Marc de treball i conceptes previs

5.1 GPU

La unitat de processament de gràfics o GPU, en termes simples, és un coprocessador. Es tracta d'un component molt semblant a la CPU, només que el tipus de processament al qual es dedica és al de gràfics. D'aquesta manera, la GPU pot alleugerir la càrrega d'informació que ha de ser processada per la unitat central, i aquesta última pot funcionar de manera més eficient. És un coprocessador especialitzat per als gràfics.

La diferència entre CPU i GPU la trobem en l'arquitectura. Tot i que estan dissenyades per funcionar de manera molt similar, les GPU estan construïdes de manera que siguin molt més eficients per al càlcul d'informació gràfica. Això les fa estar molt més optimitzades que un processador convencional per al tipus de labor en què es basen, però, no són tan bones a l'hora de dur a terme altres tasques.

Val la pena discutir breument com la GPU arriba a tant alt rendiment. El secret està en el paral · lelisme massiu.

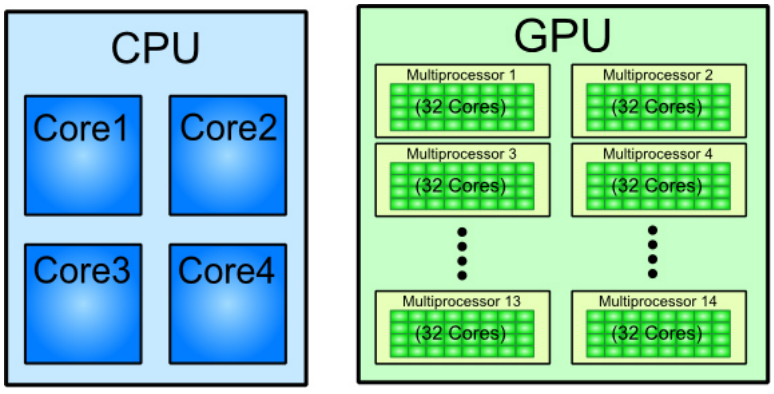


Figura 5.1: Comparativa de l'arquitectura entre CPU i GPU

Com podem veure a la Figura 5.1, una CPU té normalment entre 2 i 8 nuclis amb un rellotge d'entre 2-4GHz, mentre que una GPU té centenars de nuclis relativament simples en quan a la velocitat de rellotge (1GHz). Es poden observar grans beneficis si executem una tasca paral · lelitzada a través dels nuclis d'una GPU. De fet, aquesta arquitectura paral · lelitzable és la raó pel qual una GPU pugui processar grans quantitats de dades ràpidament.

5.1.1 Com funciona una GPU

A diferència de la CPU, dissenyada amb pocs nuclis però altes freqüències de rellotge, la GPU sol tenir grans quantitats de nuclis de processament a freqüències de rellotge relativament baixes. En l'actualitat, la majoria dels nuclis de processament estan dirigits a dues funcions: processament de vèrtexs i de píxels.

El processament de vèrtexs és relativament senzill per a les GPU modernes, sent les que menys recursos consumeixen. En termes senzills es tracta d'obtenir la informació dels vèrtexs, prèviament calculada per la CPU, i processar el seu ordenament espacial, rotació, i quin segment del vèrtex serà gràficament visible, per així continuar amb el pixelat.

A continuació es procedeix a processar els píxels, o en altres paraules, els gràfics observables com a tal. Aquest és el procés més complex i que requereix més càrrega de processament, ja que s'aplicaran totes les capes i efectes necessaris per crear textures complexes i obtenir gràfics el més realistes possibles.

Finalment, un cop processada la informació gràfica, aquesta és portada a un monitor digital o analògic (en aquest últim cas, previ pas per un convertidor), segons les necessitats pròpies de l'ordinador.

5.1.2 La *pipeline* gràfica programable

Una *pipeline* programable permet un control complet de cada etapa mitjançant *shaders*, petits trossos de codi que un cop compilats, modificant alguna de les etapes de forma dinàmica. Els *shaders* de vèrtexs són programes que realitzen les transformacions dels vèrtexs i les normals, generació de coordinades de textura, i pre-càlculs d'il · luminació que normalment es calculen en l'etapa de processament geomètric. Els *shaders* de fragments o de píxels són programes que realitzen els càlculs en l'etapa de processament dels píxels de la *pipeline* gràfica i determinen exactament com s'il · lumina cada píxel, quina textura s'hi aplica, i si el píxel s'ha de dibuixar o no.

La tendència dominant en el disseny de targetes gràfiques és posar més esforços en la programabilitat de la GPU. La Figura 5.2 mostra les etapes de processament del *vertex* i el *fragment shader* dins la *pipeline* d'una GPU programable.

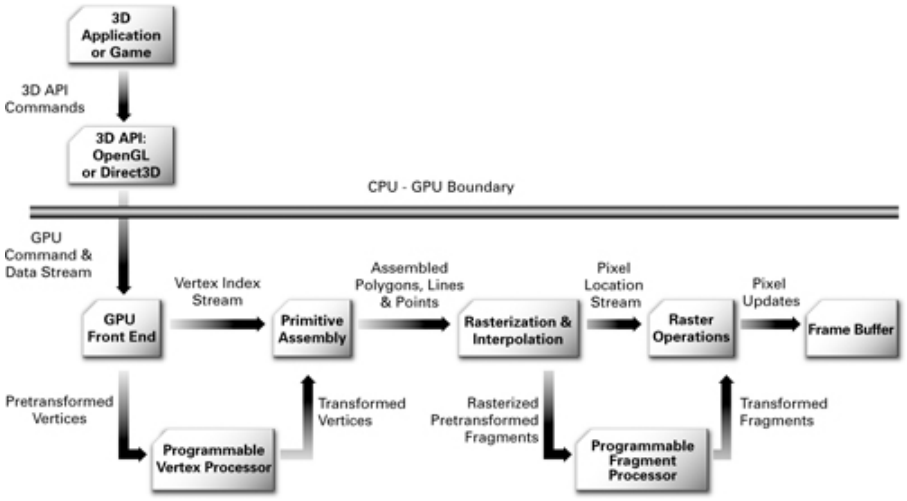


Figura 5.2: La *pipeline* gràfica programable

5.1.3 El processador de vèrtexs programable

El processador de vèrtexs s'encarrega d'executar els *vertex shaders* o programes de vèrtexs sobre el flux de dades que rep. És la primera fase programable de la *pipeline* i correspon a les primeres etapes de la *pipeline* clàssica, entre les que destaquen la transformació i il · luminació de vèrtexs. No obstant això, les GPUs programables poden realitzar més accions, incloent les relacionades amb la programació d'algoritmes genèrics.

El flux de vèrtexs que arriba al processador de vèrtexs conté dades com ara la posició, color, coordenades de textura, etc. El processador de vèrtexs executa el nucli actiu, és a dir, el *vertex shader* sobre cada vèrtex del flux de forma paral · lela. Cada vèrtex circula per una via en ser processat, per la qual cosa s'opera sobre tants vèrtexs com vies hi ha disponibles. El resultat del programa de vèrtexs només és de lectura, i passa a la següent fase de la *pipeline*. Aquest resultat inclou la nova posició del vèrtex i la modificació dels atributs del mateix. Aquests valors són utilitzats en l'etapa de rasterització (no programable) per interpolar-los en cada primitiva i obtenir els atributs dels fragments corresponents, que seran enviats al processador de fragments.

5.1.4 El processador de fragments programable

El processador de fragments executa els *fragment shaders* o programes de fragments sobre el flux de dades d'entrada. Aquest processador compte, més o menys, amb les mateixes capacitats de càlcul matemàtic que el processador de vèrtexs. No obstant això, en estar orientat a la generació de píxels, ofereix molta més flexibilitat en el tractament de textures. En aquesta etapa es permet un gran quantitat d'operacions de lectura de textures, incloent mètodes d'interpolació i filtrat.

Les GPUs modernes ofereixen suport per a operacions en coma flotant en tota la *pipeline*, i en la fase de processament de fragments es poden utilitzar textures i *framebuffers*. Cal recordar que no tots els fragments acaben generant un píxel visible en la imatge final, ja que una bona part és descartada en el procés.

5.2 OpenGL

OpenGL (*Open Graphic Library*) és una interfície de programació d'aplicacions (API), la qual merament és una llibreria per accedir a les característiques del *hardware* gràfic. Les comandes que conté permeten especificar objectes, imatges i les operacions necessàries per a produir aplicacions 3D interactives.

OpenGL s'ha dissenyat per ser eficient, independent del *hardware*, permetent així poder ser implementat en diferents tipus de targeta gràfica, o totalment en *software*. OpenGL no incorpora funcions per realitzar "tasques de finestra" o per processar l'*input* de l'usuari, tot i que les aplicacions necessitaran les funcionalitats del sistema de finestres on s'executaran. En OpenGL s'han de construir els objectes 3D a partir d'un grup de primitives geomètriques, com són els punts, línies o triangles.

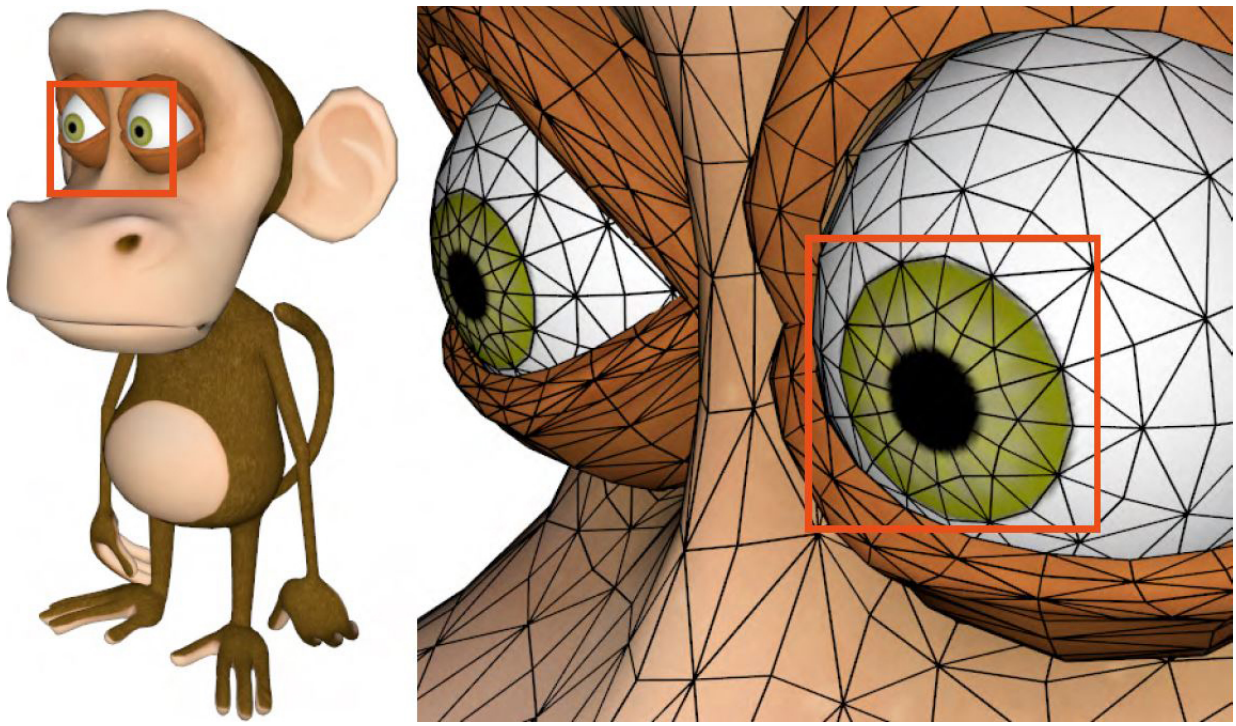


Figura 5.3: State of the Art on Real-time Rendering with Hardware Tessellation

La següent llista descriu la majoria d'operacions que una aplicació d'OpenGL hauria de realitzar per renderitzar[1] una imatge.

- Especificar les dades per construir les figures a partir de les primitives geomètriques d'OpenGL.
- Executar *shaders* per calcular, a partir de les primitives d'entrada, la seva posició, color i d'altres atributs de renderitzat.
- Convertir la matemàtica de les primitives en els respectius *fragments* associats amb la posició a pantalla. Aquest procés s'anomena *rasterització*.
- Finalment, executar el *fragment shader* per a cada fragment generat a la rasterització, que determinarà el color i posició final del fragment.
- Probablement realitzar operacions "per-fragment" addicionals, com ara determinar si l'objecte inicial és visible, o barrejar el color del fragment amb el color actual a la posició de pantalla.

[1] El renderitzat és el procés de generar una imatge des d'un model 2D o 3D (o models que formen una escena).

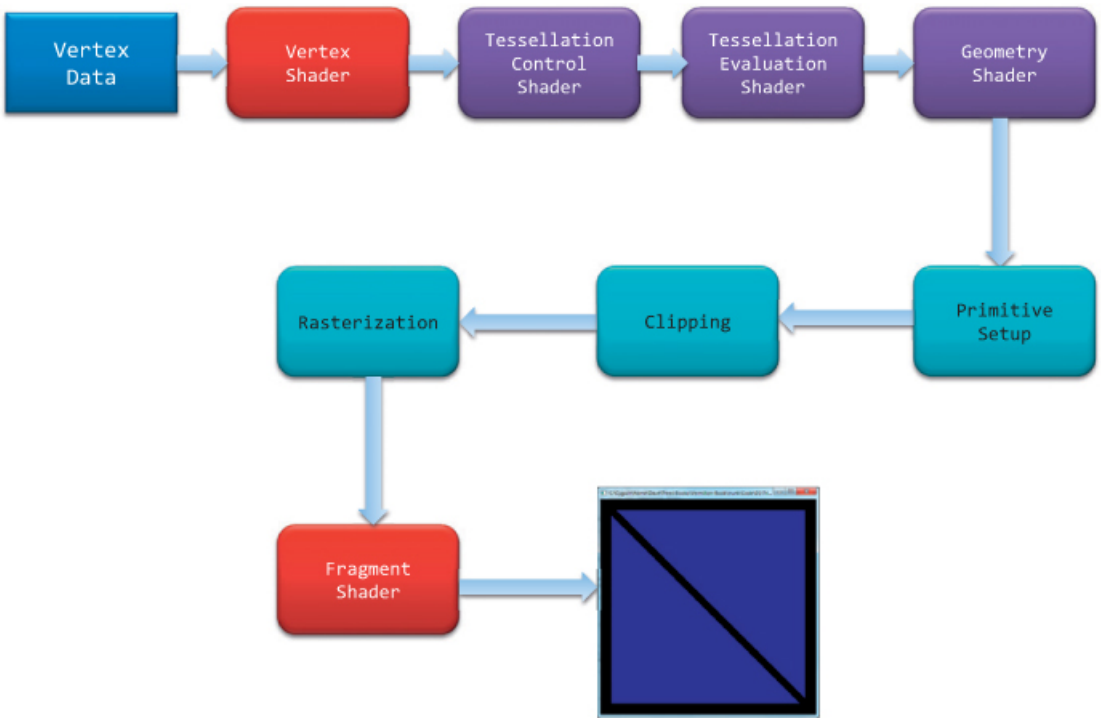


Figura 5.4: OpenGL *pipeline* associada a la Versió 4.3

Podem veure-ho més clarament amb l'exemple de la Figura 5.5.

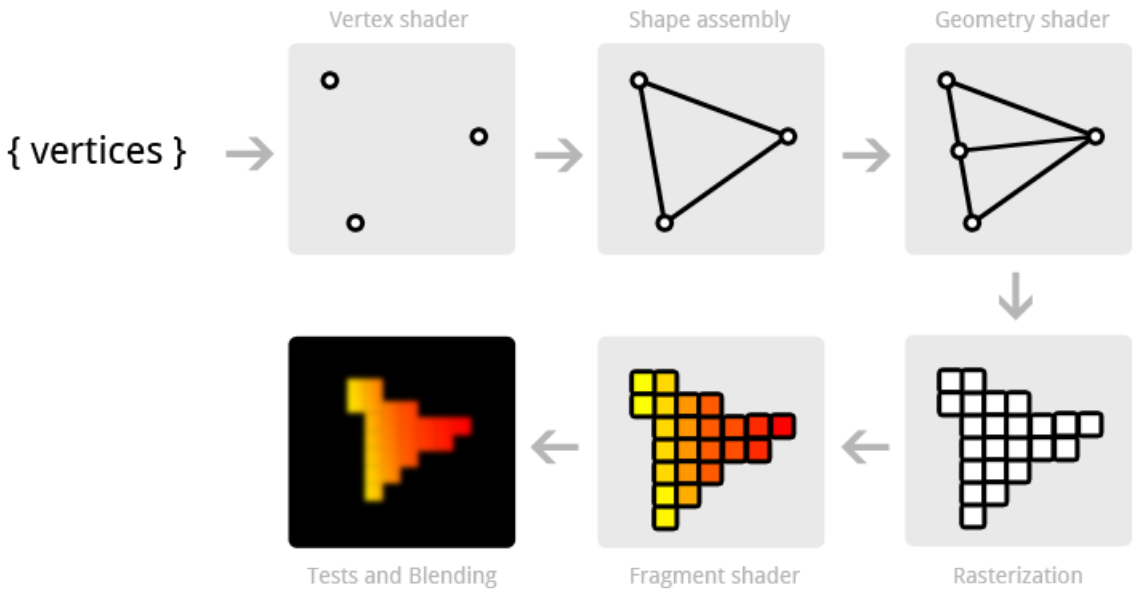


Figura 5.5: Exemple de renderitzat de 3 vèrtexs

Seguint l'exemple de la figura anterior, veiem que per arribar a dibuixar un triangle el primer que s'ha de fer és definir-ne els vèrtexs. Si el que volguéssim fos dibuixar una figura més complexa, l'hauríem de sub-dividir en varis triangles (Figura 5.3) i fer el mateix. Per a cada vèrtex del triangle, també n'hauríem de definir el color.

Per definir els vèrtex amb color, podem fer-ho de la següent manera:

```
glColor3f(red,green,blue);
glVertex3f(coordX,coordY,coordZ);
```

El vèrtex vindrà donat per les coordenades i el color per un codi de colors RGB. En el cas de l'exemple amb figures més complexes, guardarem els vèrtexs en *buffers*, és a dir, tindrem un buffer amb tots els vèrtexs de la figura, un altre *buffer* per definir les cares (agrupació dels vèrtex per índex) i un *buffer* pels colors. Per exemple, si tenim una *array* de 9 punts anomenada *points* i volem col · locar-ho en un *buffer*, farem:

```
GLuint points_vbo = 0;
glGenBuffers(1,&points_vbo);
glBindBuffer(GL_ARRAY_BUFFER,points_vbo);
glBufferData(GL_ARRAY_BUFFER,9*sizeof(float),points,GL_STATIC_DRAW);
```

Veiem que el *buffer* s'anomena *points_vbo* i se li ha d'indicar que contindrà 9 *floats* (3 per cada vèrtex). GL_STATIC_DRAW significa que aquest valors no canviaran quan s'hagin pintat. Finalment per pintar els triangles formats a partir dels vèrtex indexats en el *buffer* de cares farem:

```
glDrawElements(GL_TRIANGLES,3,GL_UNSIGNED_SHORT,BUFFER_OFFSET(0));
```

A la comanda li diem que pintem triangles, en aquest cas 1 (3 vèrtexs), on els vèrtex són números sense signe i l'*offset* apuntant a la primera posició del *buffer* d'índexs.

5.3 OpenGL Shading Language

Com hem pogut veure a la Secció 5.1.2 sobre la programabilitat de la *pipeline* gràfica, es poden programar dos unitats anomenades processador de vèrtexs i processador de fragments. En aquesta secció veurem com a partir del llenguatge OpenGL Shading Language (GLSL) podrem programar els anomenats *vertex* i *fragament shaders* per modificar aquestes unitats.

El GLSL, també conegut com GLslang, és una tecnologia part de l'API estàndard d'OpenGL, que permet especificar segments de programes gràfics que seran executats sobre la GPU (*shaders*). La contrapartida és el HLSL de DirectX. GLSL és un llenguatge d'ombrejat d'alt nivell basat en el llenguatge de programació C. Originalment introduït com una extensió d'OpenGL 1.4, GLSL es va incloure formalment en el nucli d'OpenGL 2.0 per OpenGL ARB. Va ser la primera gran revisió a OpenGL des de la creació d'OpenGL 1.0 el 1992.

Anomenem shader d'OpenGL a un programa escrit en llenguatge GLSL que es pot afegir a la *pipeline* de processament i permet afegir noves funcionalitats. Hi ha dos tipus de *shaders*, els *vertex shaders* i els *fragment shaders*, encara que en general quan parlem de *shader* ens estem referint a una combinació de tots dos, que mitjançant la realització de còmputs sobre vèrtexs i sobre fragments respectivament ens proporcionaran la manera de programar funcions pròpies per augmentar les capacitats del cicle de producció de gràfics.

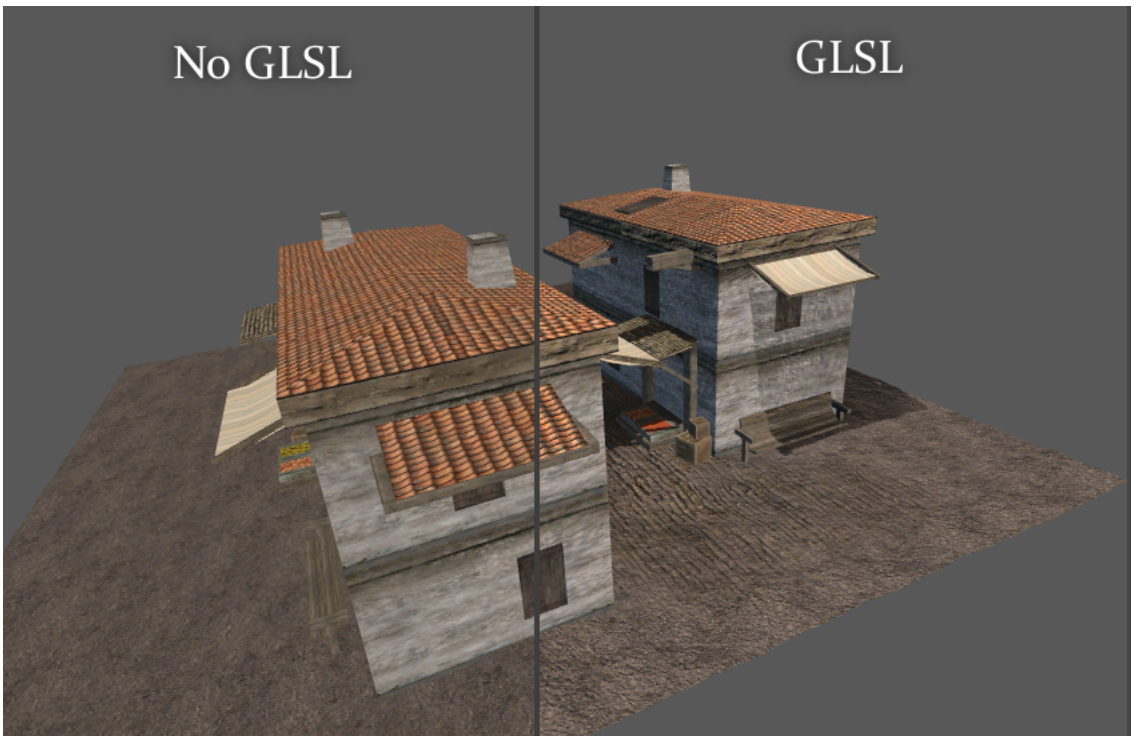


Figura 5.6: Diferència d'una mateixa escena amb i sense GLSL

5.3.1 Estructura dels *shaders*

Com s'ha dit anteriorment, el *vertex shader* s'encarrega de transformar els vèrtexs. Per a fer-ho s'han de tenir en compte dues matrius: la *ModelView matrix* i la *Projection matrix*.

La *ModelView matrix* és la concatenació de la *Model matrix* i la *View matrix*. La *Model matrix* defineix la posició en un determinat *frame* de la primitiva que es vol dibuixar, en canvi la *View matrix* defineix la posició (localització i orientació) de la càmera de l'escena. Per altra banda, la *Projection matrix* defineix les característiques d'aquesta càmera, com ara els *clip planes*, el *field of view* (fov), etc. Així, un exemple senzill de càlcul de la posició d'un vèrtex seria:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
void main(){
  gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

Per altra banda el *fragment shader* calcula el color que haurà de tenir un píxel determinat en l'escala RGBA. Un exemple molt simple podria ser:

```
void main(){
  gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Això són només alguns conceptes, a partir d'aquí es pot ampliar molt el càlcul. Més endavant veurem la utilització dels *uniforms* i les textures a partir de *sampler2D*.

5.3.2 Carregar *shaders*

Els *shaders* han de ser llegits des d'un fitxer font contingut en el disc dur, amb extensió *.vert* pel *vertex shader* i *.frag* pel *fragment shader*. En el següent codi d'exemple tenim dos punters *vs* i *fs* que apuntaran a aquests fitxers respectivament i els carregarem com a *shaders*:

```
char *vs,*fs;
GLuint v,f,p;
v = glCreateShader(GL_VERTEX_SHADER);
f = glCreateShader(GL_FRAGMENT_SHADER);
p = glCreateProgram();
```

Llegim els fitxers *.vert* i *.frag* amb la comanda *TexFileRead* i en col · loquem el contingut en el *vertex* i *fragment shader* respectivament:

```
vs = TextFileRead(vertex_shader_ruta);
fs = TextFileRead(fragment_shader_ruta);
glShaderSource(v,1,&vs,NULL);
glShaderSource(f,1,&fs,NULL);
free(vs);free(fs);
```

Compilem el *vertex* i el *fragment shader*:

```
glCompileShader(v);
glCompileShader(f);
```

Finalment enllacem el *vertex* i el *fragment* en un únic *shader*:

```
glAttachShader(p,f);
glAttachShader(p,v);
glLinkProgram(p);
```

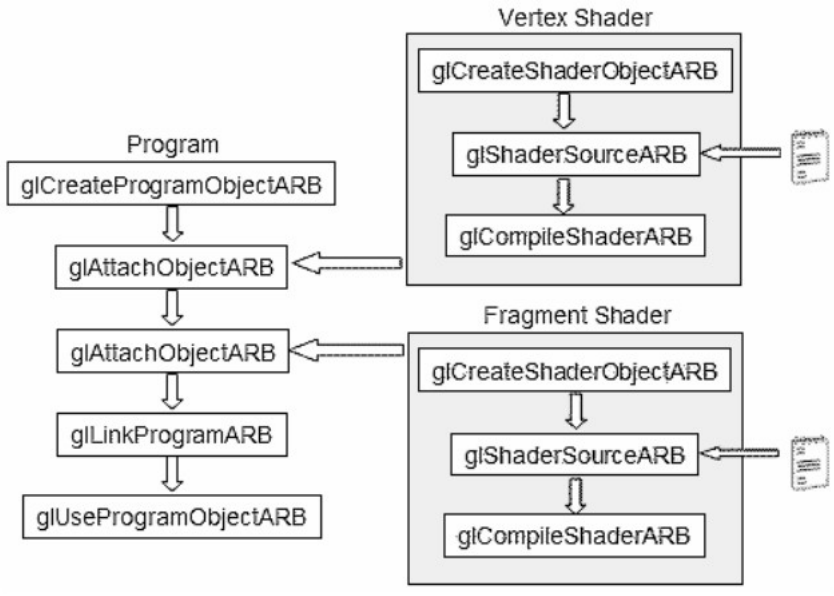


Figura 5.7: Esquema del procés de càrrega d'un *shader*

5.3.3 Les variables *uniforms*

Les variables *uniform* s'utilitzen per declarar variables globals, el valor de les quals es el mateix durant tot el processament de la primitiva. Totes les variables *uniform* són només de lectura i s'inicialitzen externament, ja sigui en temps d'enllaç o a través de l'API. El valor inicial del temps d'enllaç és o bé el valor de l'inicialitzador de la variable, si existeix, o 0 si no existeix l'inicialitzador.

Exemples de declaracions d'*uniforms* poden ser:

```
uniform vec4 lightPosition;
uniform vec3 color = vec3(0.7, 0.7, 0.2); // value assigned at link time
```

Les variables *uniform* poden ser usades amb qualsevol tipus bàsic de dada, o quan es declarara una variable on el tipus és una estructura, o una *array* de qualsevol d'aquests. Per passar una variable *uniform* a un *shader* es fa mitjançant la comanda *glUniform*:

glUniform{1|2|3|4}{f|i}, per exemple glUniform1i('posició de l'*uniform*', *valor*), on aquest *valor* és un enter (1i).

glUniform{1|2|3|4}{f|i}v, per exemple glUniform2fv, igual que l'anterior però passem un *vector* amb 2 *floats*.

glUniformMatrix{2|3|4}fv, igual que l'anterior però s'usa per passar *mat2*, *mat3*, *mat4*, o *arrays* de matrius.

5.3.4 Les textures

Les textures en GLSL es representen amb el tipus *sampler*, les quals han de ser *uniforms*. Cada *sampler* d'un programa representa una textura d'un tipus. En aquest treball s'utilitzaran dos tipus de *sampler*: *sampler2D* i *samplerCube*. El *sampler2D* és una textura de dimensions potència de 2 (normalment imatges de 64x64) i el *samplerCube* és una agrupació de 6 *sampler2D* que formen un cub.

Veiem un exemple de comunicació entre el codi d'OpenGL amb la textura del nostre *shader*:

Carreguem la textura des d'OpenGL:

```
void init2DTexture(GLint texName, GLubyte* image){
  glBindTexture(GL_TEXTURE2D, texName);
  glTextParameteri(GL_TEXTURE2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
  glTextParameteri(GL_TEXTURE2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
  glTextParameteri(GL_TEXTURE2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  glTextParameteri(GL_TEXTURE2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  glTexImage2D(GL_TEXTURE2D,0,GL_RGBA,GL_RGBA,GL_UNSIGNED_BYTE,image);
}

init2DTexture(someTexName,image);
glActiveTexture(GL_TEXTURE0); // Activem la texture pel canal 0
glBindTexture(GL_TEXTURE2D,someTexName);
location = glGetUniformLocation(shaderProgram,"myTexture"); // Localitzem la textura dins ↩
    del shader
glUniform1i(location,0); // Passem la textura al shader com a uniform
```

El que hem fet ha estat activar el canal de textura 0 (OpenGL té fins a 16 canals) i li hem associat la textura. A continuació, mitjançant la comanda *glUniform*, li diem a l'objecte GLSL que la variable del *shader* "myTexture" utilitzarà el canal 0, que és on hi hem assignat la textura.

Finalment el *fragment shader* rebrà la textura com a uniform, i a partir de les coordenades de textura rebudes des del *vertex shader* pot calcular el color final del fragment:

```
uniform sampler2D myTexture;

void main(){
  vec4 color = texture2D(myTexture,gl_TexCoord[0].st);
  gl_FragColor = color;
}
```

Pel cas dels *samplerCube* funciona exactament igual, però en comptes de passar una textura, se'n passen 6 de juntes (indicant en quina posició del cub se situa cadascuna). Llavors en el *fragment shader* en comptes d'utilitzar *texture2D* s'utilitza *textureCube*, el qual espera un *samplerCube* com a paràmetre i un *vec3* en comptes de *vec2* per a les coordenades de textura.

5.3.5 *Render to texture*

Renderitzar a una textura (*render to texture*) és una tècnica molt útil per a programació amb OpenGL. Es tracta de "pintar" l'*output* d'un *shader* GLSL a un *buffer*. Si ja hem vist els *vertex buffer object* (VBO), per guardar els vèrtexs d'una figura complexa, en aquest cas s'utilitzen *frame buffer object* (FBO), on el *fragment shader* es "pinta" en una textura i no a pantalla.
Per crear un FBO ho farem utilitzant la següent comanda:

```
void glGenRenderbuffers(GLsizei n,GLuint* ids)
```

On 'n' és el nombre de FBO que volem crear i 'ids' l'*array* que conté els noms. Per utilitzar-lo ho farem amb la comanda *glBindFramebuffer*. Veiem-ne un exemple:

```
GLuint fboId;
glGenFramebuffers(1,&fboId);
glBindFramebuffer(GL_FRAMEBUFFER,fboId);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D,GL_TEXTURE_2D, ↩
    textureId,0);
```

Finalment en el moment de pintar, en comptes de fer-ho a pantalla ho farem dins el FBO, per això haurem de cridar la funció *draw()* del programa dins el *bind* del FBO:

```
...
// Activem la renderitzacio al FBO
glBindFramebuffer(GL_FRAMEBUFFER,fboId);

// Netejem el buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// "Pintem" l'escena al FBO
draw();

// Desactivem la renderitzacio al FBO
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

En aquest moment el nostre FBO ja conté el contingut del renderitzat del programa en forma de *textura*. El que es pot fer ara és, per exemple, utilitzar aquesta textura per mapejar-la a un objecte, com s'ha dut a terme en aquest projecte i que veurem més endavant.

5.4 Projection mapping

El *projection mapping* el podem trobar en el nostre dia a dia, per exemple quan fem una projecció d'unes diapositives a la lona de les aules de la universitat, però s'intenta anar més enllà projectant imatges en qualsevol superfície, convertint qualsevol objecte 3D en pantalles interactives.

Més formalment, *projection mapping* és "una visualització d'una imatge en una superfície complexa arbitraria". També se l'anomena "video mapping" o "spatial augmented reality".

El *projection mapping* pot ser usat per publicitat, concerts, teatre, *gaming*, computació, decoració, etc.

Figura 5.8: Exemples teòrics de *projection mapping*

Cal crear una còpia virtual de la configuració física completa de l'objecte on es projectarà. Es crea un model virtual de la superfície de projecció, i es col · loca dins d'un entorn virtual. Les coordenades han de ser definides segons on es posa l'objecte en relació amb el projector. Amb aquests conceptes, en aquest treball se n'ha intentat fer una adaptació, important els objectes com a models 3D per fer-hi la "projecció". No s'ha tingut en compte la posició del projector, ja que és una simulació. En el moment de voler fer una projecció real s'hauria d'afegir la funcionalitat del projector.

En resum: el *projection mapping* intenta texturitzar un objecte normalment inanimat, tenint en compte la seva superfície irregular. Jugant amb els vèrtexs de la projecció i la perspectiva del projector. En aquest treball això es resol amb un *render to texture*.

Capítol 6

Requisits del sistema

En aquest capítol s'explica quina és la problemàtica que focalitza aquest treball, i com s'ha plantejat per resoldre-la, així com també explicar els requisits funcionals i no funcionals que ha de complir el sistema.

6.1 Plantejament de la problemàtica

Com s'ha vist en el Capítol 1, els VDJs professionals treballen amb programaris especialitzats i de pagament. Tot i això, si el que es vol es fer previsualitzacions de *mapping* d'una manera senzilla, seguint una *pipeline* robusta, amb una aplicació gratuïta, actualment no n'hi ha.

La idea d'aquest treball és la d'elaborar una aplicació que pugui simular amb pocs passos un "mapeig" sobre models 3D, sense haver de tenir coneixements de *projection mapping*, però si d'informàtica gràfica.

El perquè de tenir coneixements d'informàtica gràfica és perquè les projeccions es faran amb *shaders* de la plataforma 'Shadertoy' utilitzant models 3D, els quals podem adquirir-los de manera gratuïta, o bé dissenyar-los amb programari gratuït de modelatge 3D, com pot ser el Blender.

Així doncs, faríem les nostres projeccions integrant els *shaders* de Shadertoy i models 3D dins la mateixa aplicació, i aprofitant la tècnica del *render to texture* per fer el mapeig.

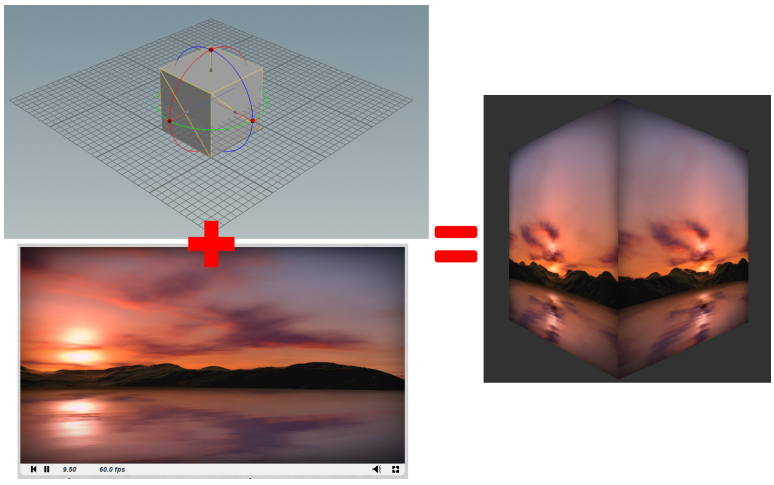


Figura 6.1: Exemple de la simulació

Per tant, seguint la Figura 6.1, la nova problemàtica a resoldre és la següent:

- Obtenir i executar de manera local els *shaders* de 'Shadertoy', amb independència del nombre i del tipus de textures utilitzades.
- Carregar models 3D obtinguts a partir de terceres persones o modelats amb programari de modelatge 3D.
- Integrar les dues funcionalitats anteriors en un entorn de treball intuitiu amb una bona usabilitat.
- Programar un *render to texture* per a simular el *projection mapping* del *shader* sobre el model 3D.

6.2 Plantejament de la solució

La solució plantejada per assolir l'objectiu del treball és:

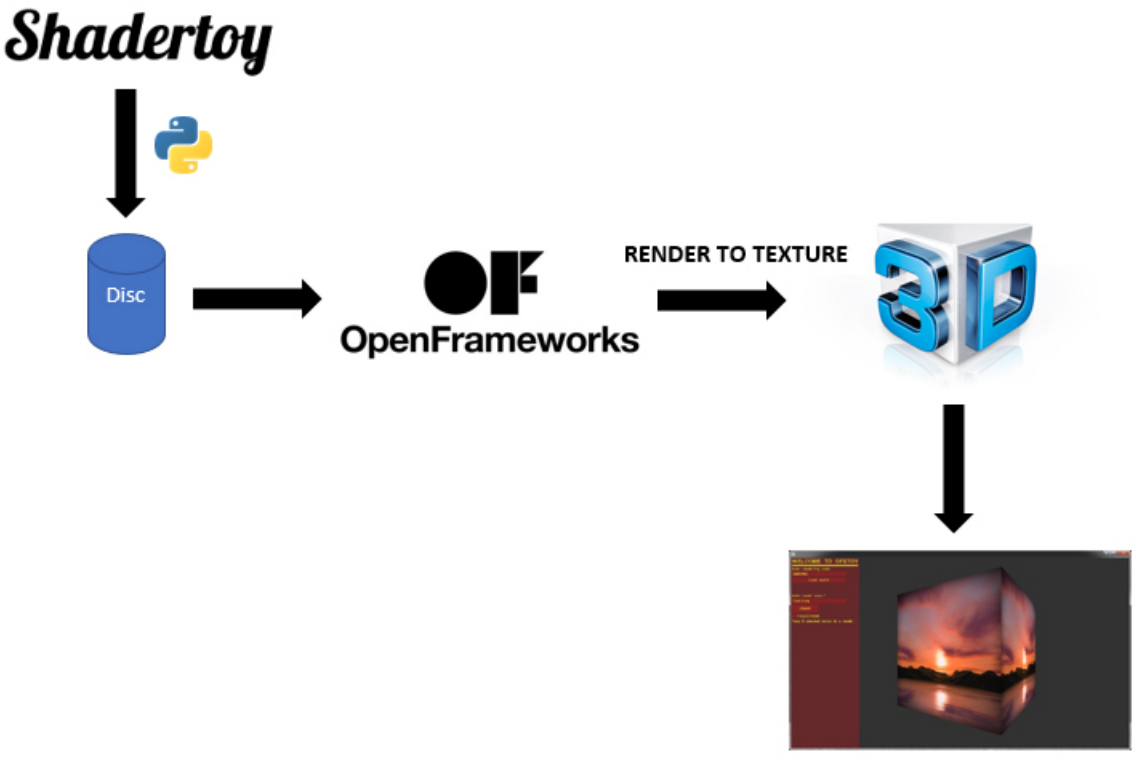


Figura 6.2: *Pipeline* de treball com a solució a la problemàtica

Com veiem a la Figura 6.2, corresponent a la *pipeline* de treball, com a solució a la problemàtica sorgida, el que haurem de fer és el següent: descarregar els *shaders* de 'Shadertoy' al disc local amb Python com a llenguatge de programació. A continuació importar els *shaders* al *framework* openFrameworks (amb independència dels *uniforms* i el nombre de textures dels *shaders*). Seguidament, si l'usuari ho demana, carregar el *shader* en un FBO mitjançant un *render to texture* i mapejar-lo en un model 3D importat, en cas contrari es "pinta" el *shader* a pantalla. El resultat final serà una aplicació amb una interfície gràfica intuïtiva, per poder veure els models 3D mapejats amb els *shaders* com a textures.

6.3 Requisits funcionals

Els requeriments funcionals expliquen què ha de fer l'aplicació, és a dir, totes les funcionalitats que tindrà sense especificar com es farà. Aquestes són les funcionalitats que tindrà l'aplicació:

- L'usuari no necessàriament ha de tenir coneixements de llenguatge GLSL, ja que pot fer les simulacions utilitzant els *shaders* d'altres usuaris de 'Shadertoy'. Ara bé, si el que vol és elaborar els seus propis *shaders*, haurà de tenir aquests coneixements i programar-los a la plataforma 'Shadertoy'.

- Un cop l'usuari sap quin *shader* vol utilitzar per a la projecció, n'hi haurà prou en apuntar el codi ID del *shader* a l'aplicació perquè aquesta el descarregui al disc local amb tot el necessari (aplicació Python).

- Compilació i execució dels *shaders* descarregats per part d'openFrameworks.

- L'aplicació ha de permetre "mapejar" els *shaders* de 'Shadertoy' si l'usuari vol fer una previsualització de *mapping*.

- Funcionalitat *render to texture* en openFrameworks si l'usuari vol mapejar el *shader* com a textura a sobre d'un model 3D.

- Lliure accés al model utilitzant ratolí i tecles WASD per poder moure i veure el model des de tots els punts i perspectives.

- Una interfície d'usuari intuïtiva, colorida, per poder entrar el codi ID del *shader*, escollir entre mapejar a pantalla o al model 3D, seleccionar el model 3D entrant només el nom, possibilitat d'executar-ho a pantalla completa.

6.4 Requisits no funcionals

En aquest apartat es defineix com ha de ser l'aplicació, i no què és el que ha de fer. Aquests requisits fan referència al que s'ha de tenir en compte a l'hora de dissenyar el sistema, com per exemple els recursos necessaris.

Els requisits no funcionals són els següents:

- Les tasques de descàrrega dels *shaders* i compilació i execució d'aquests s'ha de fer amb el mínim temps possible, que no suposi una càrrega d'espera gran per a l'usuari. S'ha de tenir em compte que aquest temps dependrà de la connexió a Internet que l'usuari disposi a l'hora de descarregar els *shaders*.

- Els perifèrics necessaris per a la bona utilització de l'aplicació seran els bàsics: ratolí i teclat.

- Els requisits de *hardware* mínims són: sistema operatiu Windows 7, 4Gb de memòria RAM, 1Gb d'espai lliure en el disc, targeta gràfica amb suport DirectX 10 i *drivers* actualitzats, OpenGL actualitzat, teclat i ratolí.

Aquest treball ha estat provat sota una targeta gràfica amb DirectX 11.

Capítol 7

Estudis i decisions

En aquest capítol descriurem el programari i les llibreries utilitzades durant el desenvolupament del projecte, i en justificarem l'elecció. En el cas de les llibreries, n'explicarem el funcionament bàsic per poder comprendre-les.

7.1 Shadertoy

'Shadertoy' és la primera aplicació, des del 2009, en permetre als desenvolupadors de tot el món "pintar" píxels a la pantalla usant WebGL [1].

És una pàgina web que parteix d'aquesta idea original de WebGL. D'una banda, s'ha reconstruït amb la finalitat de proporcionar als desenvolupadors de gràfics per ordinador i aficionats, una gran plataforma per crear prototips, experimentar, ensenyar, aprendre, inspirar i compartir les creacions amb la comunitat. De l'altra, l'expressivitat dels *shaders* ha sorgit en permetre diferents tipus d'*inputs*, com ara textures.

Algunes de les característiques més destacades són:

- Codificació i compilació de *shaders* en temps real.
- Una comunitat forta de desenvolupadors de *shaders*.
- Discussió oberta amb la possibilitat de comentar i fer *like*.
- Inputs dels *shaders*: textures, temps, ratolí, vídeos, etc.
- Vista dels *shaders* a pantalla completa.
- Editor de text amb sintaxi destacada.
- Pujar i publicar els teus millors *shaders*.

[1] WebGL és una API per *JavaScript* basada en OpenGL ES 2.0 per renderitzar gràfics en 2D i 3D en qualsevol navegador web, sense necessitat de *plug-ins*

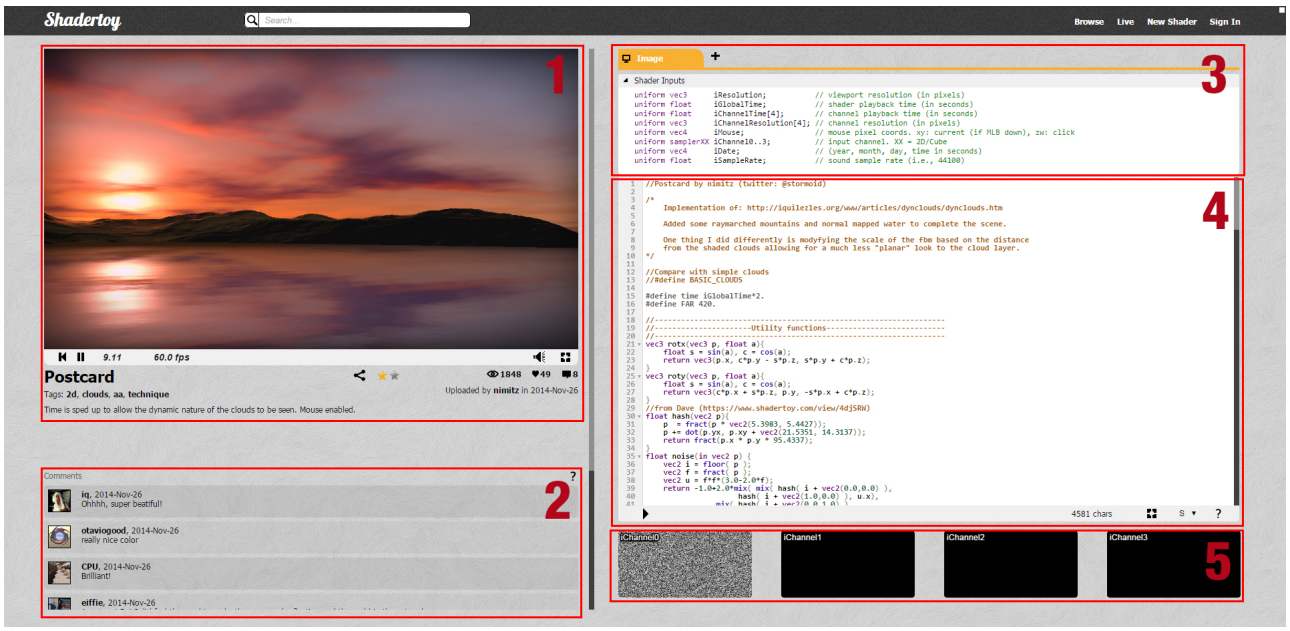


Figura 7.1: Vista de l'entorn web de 'Shadertoy'

Com podem veure a la Figura 7.1, dins l'entorn web de 'Shadertoy' ens trobem amb 5 seccions diferents:

1. Pantalla on podem veure el resultat del codi GLSL, és el *shader*. A part, podem trobar el títol i descripció del *shader*, l'*username* de l'autor i el nombre de visualitzacions, *likes* i comentaris que han fet altres usuaris.

2. Secció on podem veure els comentaris que han fet alguns usuaris respecte del *shader*.

3. *Inputs* que pot tenir el *shader*. Són els *uniforms*. Més endavant veurem com els haurem de passar des del frameworks.

4. Editor de text. Aquesta secció és per codificar el nostre *shader*, o bé modificar-ne de ja fets. Aquest codi acabarà sent el *fragment shader*, ja que com veurem 'Shadertoy' sempre utilitza el mateix *vertex shader*.

5. Canals de textura per utilitzar com a *sampler2D* o *samplerCube*. Com a màxim poden haver-hi 4 textures simultàniament.

Els *shaders* que utilitzarem han d'implementar la funció *mainImage()*, que serà cridada des del motor principal del *shader* (*main*) per poder-lo executar, per tal de generar imatges procedurals a partir del comput del color de cada píxel. Aquesta funció serà cridada per cada píxel per obtenir el color com a *output* corresponent als píxels de pantalla. La capçalera és:

```
void mainImage( out vec4 fragColor, in vec2 fragCoord );
```

On *fragCoord* conté les coordenades del píxel el qual el *shader* n'ha de computar el color. Les coordenades estan en píxels com a unitat de mesura, que van de 0.5 a resolució-0.5, sobre la superfície del *render*, on la resolució és passada com a *uniform* utilitzant la variable *iResolution*.

El color resultant es troba a la variable *fragColor* com a *vec4*.

La resta d'*uniforms* que pot rebre un *shader* com a *input* són:

```
uniform vec3 iResolution;
uniform float iGlobalTime;
uniform float iChannelTime[4];
uniform vec4 iMouse;
uniform vec4 iDate;
uniform float iSampleRate;
uniform vec3 iChannelResolution[4];
uniform samplerXX iChanneli;
```

La majoria de *shaders*, però, no utilitzen els *uniforms* iChannelTime, iDate, iSampleRate i iChannelResolution, per tant, aquests *uniforms* no s'han tingut en compte en aquest treball.

Com hem dit abans, l'editor de text de 'Shadertoy' serveix només per escriure el *fragment shader*, perquè el *vertex shader* és sempre el mateix:

```
attribute vec2 pos;

void main() {
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * vec4(pos.xy,0.0,1.0);
}
```

Com es pot observar, només es transforma la posició del vèrtex de l'espai virtual 3D, en la posició del sistema de coordenades 2D corresponents a la pantalla. La diferència és que aquí no s'utilitza el *vec4 gl_Vertex*, sinó que només s'agafen les coordenades *xy* i les altres dues són fixes.

Un cop programat el *fragment shader* i sabent que el *vertex shader* ja el proporciona internament el 'Shadertoy', dins el motor d'aquest hi ha un *Javascript* que munta el *shader* depenent dels *inputs* dels 4 canals. Amb això, bàsicament el que fa es construir el *fragment shader* final de la següent manera:

1. Munta un *string* amb tots els *uniforms* de base, i hi concatena un altre *string* amb els *uniforms* dels canals. Aquesta seria la part de la capçalera del *fragment shader*.
2. A continuació hi concatena, també com a *string*, el text de l'Editor de text, que és on es construeix la funció *mainImage*.
3. Seguidament, es concatena la funció *main()* principal, que és la següent:

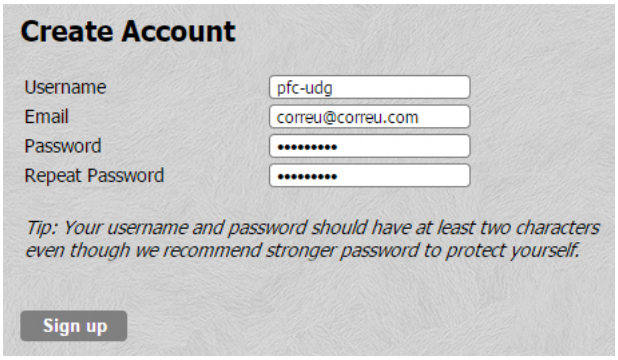
```
void main( void ){
    vec4 color;
    mainImage( color, gl_FragCoord.xy );
    color.w = 1.0;
    gl_FragColor = color;
}
```

4. En aquest moments el 'Shadertoy' ja té muntat el *fragment shader* en un *string*. Com que el *vertex shader* és sempre el mateix, també el té guardat per defecte en un altre *string* i ja només li queda compilar.
5. Per compilar-ho utilitza la mateixa estratègia explicada a la secció 5.3.2, amb comandes molt semblants però de WebGL, i en comptes de llegir els *shaders* de fitxer ho fa a partir dels *strings*. Ja només li queda linkar-los per crear un *shader* únic per poder-lo executar.

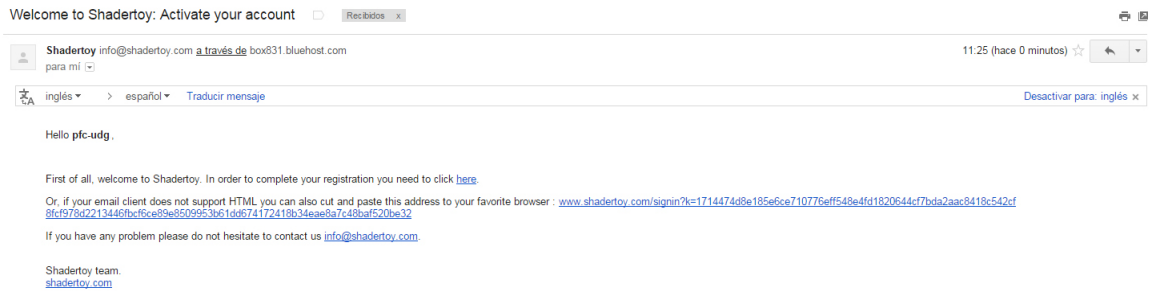
7.1.1 Com registrar-se a 'Shadertoy'

A continuació explicarem quins passos cal seguir per tal de poder-se registrar a 'Shadertoy' i començar a programar els *shaders*:

1. Accedim a donar-nos d'alta a https://www.shadertoy.com/signin i emplenem el formulari 'Create Account':

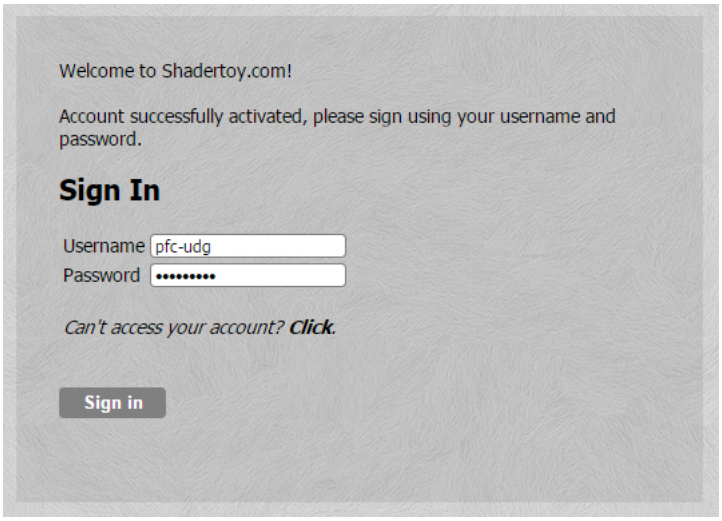


2. Un cop confirmat amb el botó 'Sign up' rebrem un correu per confirmar el registre:



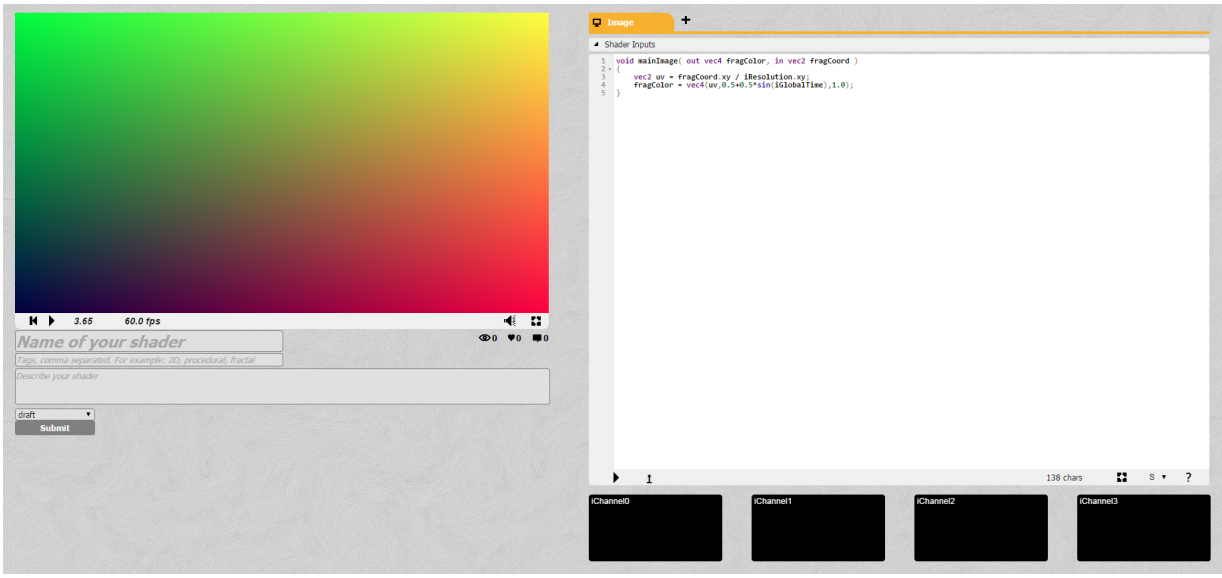
   Confirmem el registre fent clic a l'enllaç 'here'.

3. Se'ns redigirà de nou a la pàgina de 'Shadertoy' per identificar-nos amb el nou compte.



   Ens identifiquem.

4. I un cop identificats, ja podem fer clic al botó 'New Shader' del menú per començar a programar!



7.1.2 Motius de la selecció

'Shadertoy' permet programar i visualitzar, de manera molt senzilla, infinitat de *shaders*. Poden ser més o menys complicats, però només cal programar el *fragment shader* per veure'n els resultats, ja que darrera hi ha un motor que se n'encarrega de tota la resta. O bé, si es prefereix, es pot utilitzar els *shaders* d'altres usuaris.

Amb això, i el coneixement que se'n tenia de l'aplicació, el motiu de la selecció va ser la voluntat de voler executar aquest *shaders* en l'ordinador local i aprofitar-los per al *projection mapping*, ja que no hi ha cap altra aplicació que et permeti treballar amb *shaders* online.

7.2 Code::Blocks

'Code::Blocks' és un IDE (*Integrated development environment*) per a llenguatges *C*, *C++* i *Fortran*, de codi obert i multiplataforma. Usant una arquitectura de *plugins*, les seves capacitats i característiques es defineixen pels *plugins* proporcionats. Actualment, l'última versió oficial és la 13.12, però nosaltres utilitzarem la 12.11. Més endavant, a la Secció 7.2.1, explicarem el perquè d'aquesta elecció.

'Code::Blocks' no és un compilador, sinó un entorn de compilació que inclou un editor de projectes, editor de text, compilador, enllaçador i depurador. El compilador que es subministra amb 'Code::Blocks' és 'MinGW'[2].

Algunes de les característiques més important del 'Code::Blocks' són:

1. Compilador
    - Suport de nombrosos compiladors.
    - Personalització ràpida del sistema de compilació.
    - Suport per a compilació en paral · lel (utilitzant nuclis extra de la CPU).
    - Projectes *multi-target*.
    - Possibilitat de combinar projectes en el mateix entorn de treball.
    - Dependències entre projectes dins l'entorn de treball.
    - Possibilitat d'importar projectes *MSVC*[3] i *Dev-C++*[4].
2. Debugador
    - Interfícies GNU GDB (debugador estàndard del sistema operatiu GNU).
    - Suport de MS CDB (*Microsoft Console Debugger*).
    - Total suport de *breakpoints*
        - *Breakpoints* en codi.
        - *Breakpoints* en dades (lectura, escriptura i lectura/escriptura).
        - *Breakpoint conditions* (atura't només si l'expressió és *true*).
        - *Breakpoints* ignorant comptador (atura't només després de cert nombre d'accessos).

[2] 'MinGW' (*Minimalist GNU for Windows*) és una implementació dels compiladors GCC per a la plataforma *Win32*, que permet migrar la capacitat d'aquest compilador a entorns 'Windows'.

[3] *MSVC* (*Microsoft Visual C++*), actualment més conegut com Visual Studio, és l'IDE per a llenguatges *C* i *C++* de Microsoft

[4] *Dev-C++* és un IDE de codi obert per a projectes amb llenguatge *C* i *C++*

- Mostra símbols i arguments de funcions locals.
- Suport per veure els tipus definits per l'usuari durant l'encriptació.
- Crida a la pila.
- Desensemblar.
- Bolcat de memòria personalitzable.
- Possibilitat de canviar entre *threads*.
- Possibilitat de veure els registres de la CPU.

3. Interfície

- Sintaxi de colors, personalitzable i extensible.
- Plegar codi en llenguatges *C*, *C++*, *Fortran*, *XML*, entre d'altres.
- Interfície amb pestanyes.
- Terminació de codi.
- Navegador de classes.
- Guió intel · ligent.
- Intercanvi entre fitxers .h i .c/.cpp amb una tecla.
- Finestra amb la llista de fitxers.
- "Eines" externes personalitzables.
- Administració de llista "To-do" entre diferents usuaris.

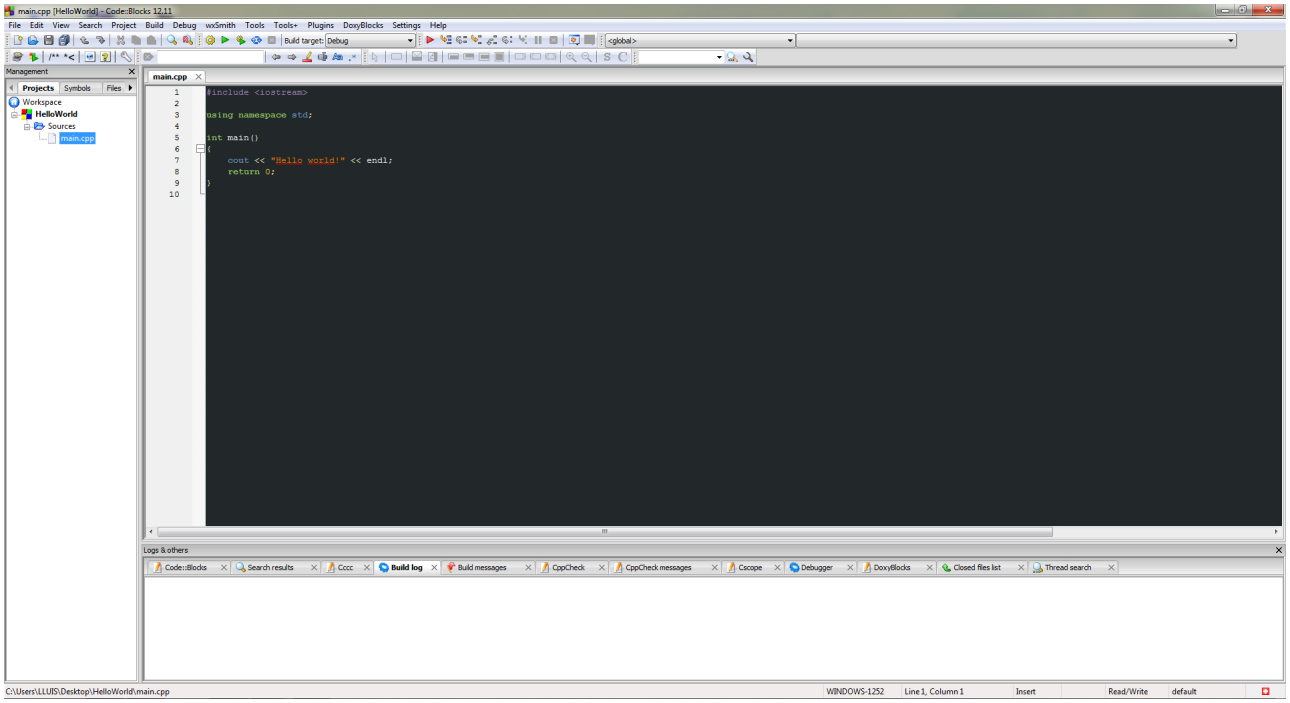


Figura 7.2: Vista general de l'entorn 'Code::Blocks'

7.2.1 Motius de la selecció

Com veurem a la següent secció, treballarem amb un *framework* (openFrameworks) per poder desenvolupar el projecte sota una eina que treballa conjuntament amb moltes llibreries, i que a més té els seus propis *addons* desenvolupats per usuaris. Aquest *framework* està desenvolupat en *C++* i suportat en diferents IDEs depenent del sistema operatiu. En el cas d'aquest treball, que treballem amb Windows, openFrameworks està suportat per *Code::Blocks* i *Visual Studio 2012*.

Podíem haver desenvolupat aquest projecte en qualsevol dels dos IDEs, però vam decantar-nos per 'Code::Blocks' pel fet de ser de codi obert, a més d'haver-lo estat utilitzant durant la carrera i tenir l'experiència del funcionament.

7.3 openFrameworks

openFrameworks és una *toolkit* de codi obert en *C++* dissenyada per assistir al procés creatiu proveint un simple i intuïtiu *framework* per a l'experimentació.

openFrameworks s'ha dissenyat per treballar conjuntament amb diverses llibreries que s'usen regularment, incloent:

- *OpenGL*, *GLEW*, *GLUT*, *libtess2* i *cairo* pels gràfics.
- *rtAudio*, *PortAudio*, *OpenAL* i *Kiss FFT* o *FMOD* per entrada, sortida i anàlisi de so.
- *FreeType* per fonts.
- *FreeImage* per guardar i carregar imatges.
- *Quicktime*, *GStreamer* i *videoInput* per reproducció de vídeo i *grabbing*.
- *Poco* per utilitats diverses.
- *OpenCV* per visió per computador.
- *Assimp* per carregar models 3D.

El codi està escrit per ser altament *cross-compatible*. En aquests moments se suporten cinc sistemes operatius (Windows, OSX, Linux, iOS i Android) i quatre IDEs (XCode, Code::Blocks, Visual Studio i Eclipse). L'API està dissenyada per ser minimalista i fàcil d'entendre.

openFrameworks està distribuït sota la *MIT License*. Aquesta et permet utilitzar openFrameworks en qualsevol context: comercial, no comercial, públic o privat, en codi obert o tancat.

7.3.1 Filosofia de disseny

openFrameworks es guia per una sèrie d'objectius: ha de ser col · laboratiu, coherent i intuïtiu, multiplataforma, de gran abast i extensible.

Col · laboratiu

El desenvolupament d'openFrameworks és col · laboratiu. Es nodreix de les aportacions de moltes persones, que es dediquen a la discussió freqüent, i col · laboren en *addons* i projectes.

Simplista

openFrameworks intenta equilibrar usabilitat i simplicitat. Les primeres versions d'openFrameworks utilitzaven el nucli com a eina per ensenyar *C++* i *OpenGL*, però amb el pas del temps els exemples s'han convertit en la millor manera d'aprendre mentre que el nucli s'aprofita de les característiques més avançades. En canvi, s'han creat molts més exemples que vénen amb openFrameworks, amb l'objectiu d'intentar fer punts de partida simples per a l'experimentació.

Consistent i intuïtiu

openFrameworks és consistent i intuïtiu: ha d'operar en el principi de la mínima sorpresa, així que el que s'aprèn d'una part d'openFrameworks es pot aplicar a altres parts del mateix. Els principiants poden usar openFrameworks per aprendre sobre els patrons comuns de programació, i els usuaris avançats poden aplicar la seva experiència en altres llenguatges i eines.

Multiplataforma

openFrameworks és una *toolkit* multiplataforma. Suporta tants entorns de desenvolupament com sistemes operatius sigui possible. Està dissenyat per treballar en multitud de plataformes: OSX, Windows, Linux, iOS, Android, sistemes ARM Linux, així com plataformes experimentals com Blackberry PlayBook. Els desenvolupadors d'openFrameworks han ideat algunes maneres enginyoses d'interactuar amb altres llenguatges, com ara Java en el cas d'Android, o Objective-C en el cas d'iOS.

Potent

openFrameworks és potent: permet aprofitar biblioteques avançades com ara OpenCV, utilitzar eficientment *hardware* com ara la targeta gràfica, i connectar perifèrics com ara càmeres i d'altres dispositius.

openFrameworks encapsula altres llibreries com són OpenGL, Cairo, FreeType, FreeImage i OpenCV. Es pot dir que openFrameworks és una capa entre el codi i l'usuari i aquestes llibreries.

Extensible

openFrameworks és extensible. És fàcil crear *addons* per estendre openFrameworks, poden ser un fragment de codi o encapsular llibreries molt més complexes com ara OpenNI, Tesseract o Box2d. El nom dels *addons* normalment comencen amb el prefix "ofx", permetent que es diferenciïn fàcilment del codi principal. A més s'inclouen "*addons* principals" que poden ser usats pels usuaris, com ara ofxOpenCV, però no són necessaris per tots els projectes.

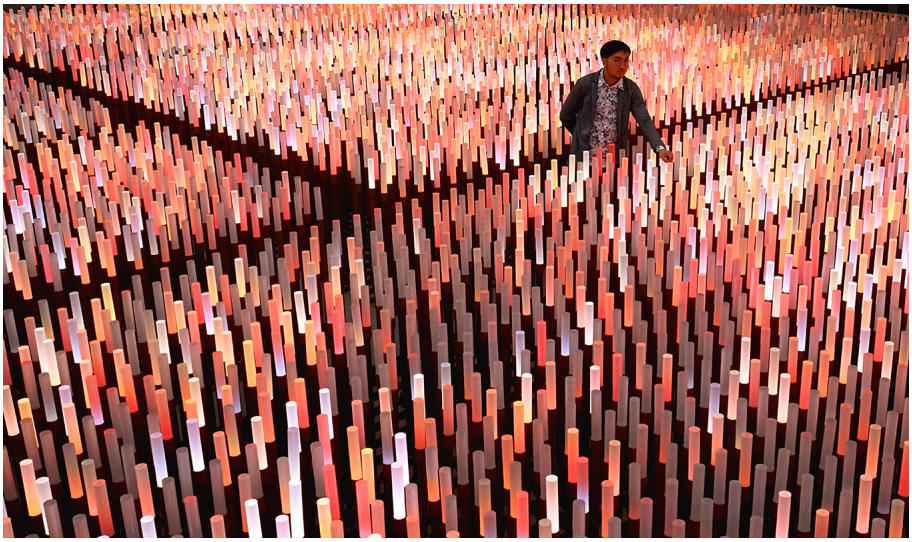
7.3.2 Projectes fets amb openFrameworks

Deserve

Deserve, dissenyat per Nick Hardeman, és un experiment visual interactiu que es troba com una aplicació OSX independent. L'aplicació comença amb un senyal audiovisual reactiu, que respon als *tracks* FFT del so.

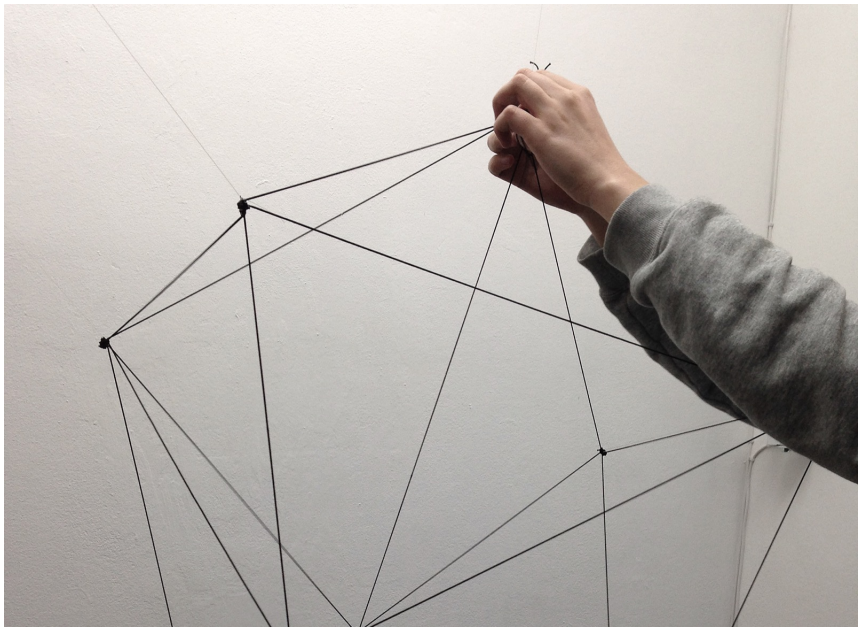
L'aplicació està construïda en openFrameworks amb models 3D animats i configurats en 'Blender' i exportats a format FBX per ser carregats en openFrameworks utilitzant l'*addon ofxFBX*. El grup de criatures segueix el comportament de congregació per simular un ramat, inspirat en el ramat Gallimimus a 'Jurassic Park'.

The Field of Hope



The Field of Hope, dissenyat per un equip de la Universitat de Tsinghua, liderat per Danqing Shi, és una instal · lació lluminosa per a la EXPO 2015 de Milan, al pavelló de la Xina. Consta de 30.000 "canyes" de metall, cadascun amb una punta difusor LED, que funcionen com un camp de visualització de píxels en 3D.

Irregular Polyhedron



Irregular Polyhedron, creat per Oslo, basat en el disseny computacional de l'estudi *Void*, és una representació física dels components bàsics de la informàtica gràfica; el vèrtex i l'aresta. La geometria és generada proceduralment (utilitzant *Rhino*, *grasshopper* i *Kangaroo*), optimitzant les dimensions de la sala, les possibles ubicacions dels motors i els vectors direccionals de tensió cap a cada vèrtex. Aquest model s'ha exportat per simulacions de física en openFrameworks i mesures per tallar i connectar cadenes de peces físiques.

Lilium

Lilium, creat per Kenichi Yoneda (Kynd) en col · laboració amb Tokyo 'visual label' BRDG, és un experiment audiovisual. El projecte es compon de molts objectes individuals d'openFrameworks que són alimentats amb dades de so transformat FFT i gràfics dibuixats en dos FBOs. Un utilitzat com a *canvas* o paper per renderitzar (en el qual, l'aquarel · la s'aplica amb *shaders*), i l'altre serveix com a *force map* utilitzat per moure els píxels en la imatge renderitzada per simulat la distorsió i moviment de l'aigua.

7.3.3 L'entorn openFrameworks

A openFrameworks sempre es treballa de la mateixa manera, a partir d'una classe *main* i d'una classe *ofApp*. Tots els objectes incorporats en openFramweorks comencen pel prefix *of*, i els creats per tercers (*addons*) *ofx*, com hem explicat anteriorment, per diferenciar-los. A la classe *main* es defineixen les dimensions de la finestra on s'executarà l'aplicació, per defecte:

```
#include "ofMain.h"
#include "ofApp.h"

//========================================================================
int main( ){
  ofSetupOpenGL(1024,768,OF_WINDOW);      // <-------- setup the GL context

  // this kicks off the running of my app
  // can be OF_WINDOW or OF_FULLSCREEN
  // pass in width and height too:
  ofRunApp(new ofApp());
}
```

Veiem que s'utilitzen dues funcions d'openFrameworks: *ofSetupOpenGL* i *ofRunApp*. *ofSetupOpenGL* crea la finestra i *ofRunApp* és la funció principal del motor d'openFrameworks, aquesta funció serà la que permetrà "pintar" com explicarem a continuació.

Per a la classe *ofApp*, s'han de definir els respectius arxius *ofApp.cpp* i *ofApp.h*, per defecte:

```
#pragma once

#include "ofMain.h"

class ofApp : public ofBaseApp{

  public:
    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y );
    void mouseDragged(int x, int y, int button);
    void mousePressed(int x, int y, int button);
    void mouseReleased(int x, int y, int button);
    void windowResized(int w, int h);
    void dragEvent(ofDragInfo dragInfo);
    void gotMessage(ofMessage msg);

};
```

Aquestes són les funcions que per defecte ha de tenir l'aplicació d'openFrameworks a *ofApp.cpp*, de les quals obligatòriament *setup*, *update* i *draw* s'han d'implementar. Aquesta obligatorietat és perquè són les tres funcions que s'executen dins la funció *ofRunApp* que es crida al *main*. El funcionament d'aquestes tres funciones és el següent:

1. *setup*: En aquesta funció s'inicialitzen els valors de les variables globals, interfície gràfica, o, en el nostre cas, carregar un *shader*, entre moltes altres possibilitats.

2. *update*: Quan es detecti un *event* a l'aplicació, capturat per les funcions no obligatòries, s'executa aquesta funció per tractar aquest *event*.

3. *draw*: Aquesta funció és l'encarregada de "pintar" els resultats a pantalla.

openFrameworks té una classe anomenada *MainLoop* on, a partir del *ofRunApp*, s'executen aquestes funcions en ordre: *setup*, *update* i *draw*. Després del primer *loop* s'estan executant les funcions *update* i *draw*, fins tancar l'aplicació.

Tota aquesta configuració inicial la podem fer de manera molt ràpida gràcies a una aplicació incorporada en el *framework*, anomenada *projectGenerator*:

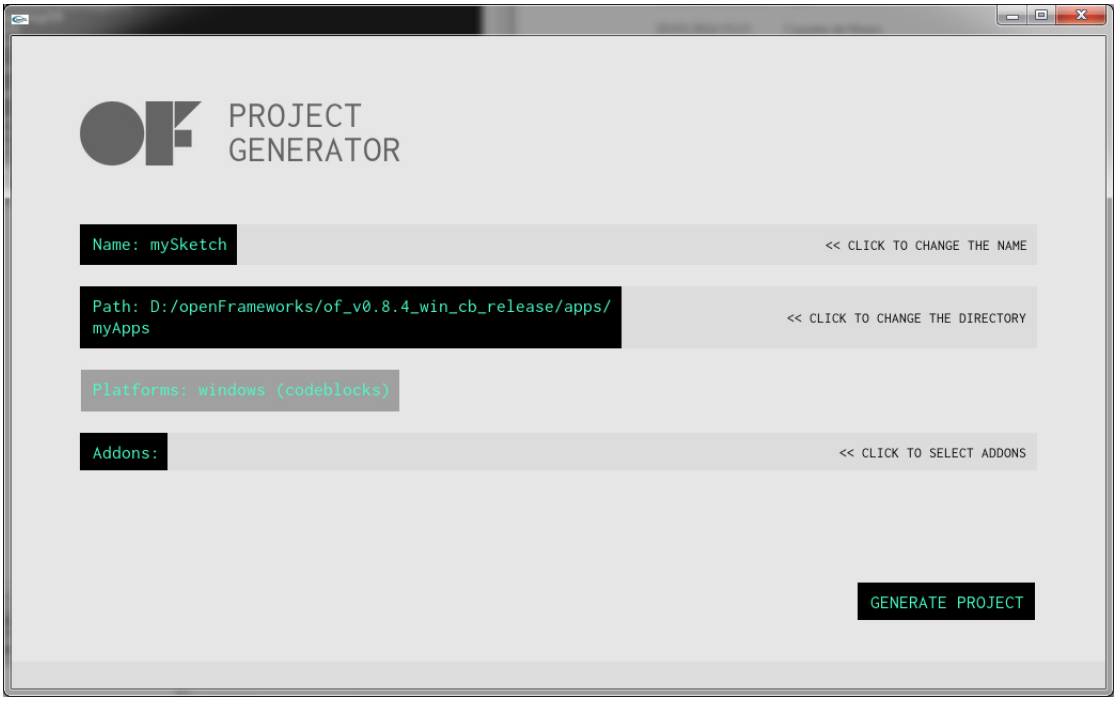


Figura 7.3: Interfície gràfica de l'aplicació *projectGenerator*

Per generar un projecte només cal indicar el nom, ubicació (per defecte carpeta /myApps) on es guardarà, plataforma i els *addons* que s'utilitzaran. Quan generem el projecte es crearà una aplicació openFrameworks amb els arxius *main* i *ofApp* per defecte, com els que hem vist, adaptat a l'IDE escollit enllaçat amb els *addons* que s'utilitzaran. Si ens interessa, com és el cas d'aquest treball, utilitzar *addons*, primer els hem de descarregar i guardar a la carpeta d'*addons* corresponent, per poder-los seleccionar des del *projectGenerator*.

7.3.4 Shaders en openFrameworks

Els *shaders* (GLSL) poden ser utilitzats a openFrameworks utilitzant l'objecte *ofShader*, que encapsula, entre d'altres, les funcions més importants de GLSL vistes a la Secció 5.3 per fer més intuïtiva la seva utilització dins del *framework*. En aquest apartat s'explicarà com utilitzar aquestes funcions per un correcte funcionament.

De la mateixa manera que s'ha explicat, hem de tenir a mà tant el *vertex* com el *fragment shader* amb les respectives extensions *.vert* i *.frag*. Per carregar i enllaçar-ho en un *shader* s'utilitza la funció *load*:

```
ofShader::load(string shaderName)
```

shaderName és el nom del *shader*, és a dir si tenim: *shader.vert* i *shader.frag*, *shaderName* = *shader*. Això vol dir que tant el *vertex* com el *fragment shader* s'han de dir igual. Interiorment

la funció *load* carrega els *shaders* a partir dels arxius, els compila i els enllaça per crear un únic *shader*, que es guardarà a la variable.

Per activar un *shader* s'utilitza la funció *begin*, per exemple *shader.begin()*, un cop activat es pot interactuar passant *uniforms*, i tot el que es pinti es veurà afectat pel *shader*. Per desactivar-lo n'hi ha prou de fer *shader.end()*.

Passant uniforms

No hi ha masses canvis en el funcionament dels *uniforms*. Per exemple, si volem passar la resolució de l'aplicació, fem:

```
float resolution[] = {ofGetWidth(),ofGetHeight(),1};
shader.setUniform3fv("iResolution",resolution);
```

On *resolution* és un vector amb la resolució configurada a la classe *main* (*ofGetWidth* i *ofGetHeight* retornen l'amplada i alçada de la finestra), i ho passem a l'*uniform iResolution* del *shader*.

Les textures

Les textures en openFrameworks les trobem a l'objecte *ofTexture*. Aquesta classe permet inicialitzar, carregar, obtenir les dades, pintar, configurar-ne el *wrap*[5], etc., d'una textura d'OpenGL.

Per defecte les textures d'openFrameworks són del tipus ARB, i per tant, no obligatòriament de dimensions potència de 2. A més, l'"origen" de coordenades del sistema es troba al vèrtex superior esquerra de la finestra, això s'ha de tenir en compte a l'hora de pintar, ja que en OpenGL aquest origen és el vèrtex inferior esquerra. Podrem veure com es tracta aquest inconvenient a la Secció **??**.

Per utilitzar textures en openFrameworks, a part de tenir en compte l'objecte *ofTexture*, també necessitarem l'objecte *ofImage* per carregar la imatge que ens farà de textura. La funció a tenir en compte serà:

```
bool ofImage::ofLoadImage(ofTexture &tex,string path)
```

Veiem-ne un exemple:

```
// ofApp.h
// ...
ofTexture tex; // Variable que actuara de textura
// ...
// ofApp.cpp
// Carreguem la imatge (.jpg per exemple) a la variable 'tex'
ofLoadImage(tex,"path");
// Passem la textura com a uniform del nostre shader
shader.setUniformTexture("myTexture",tex,0);
```

[5]Quan una coordenada de textura surt del rang 0-1, s'ha de tenir en compte com es mostra aquesta textura: *repeated* o *clamped*

Normalment el *path* sol ser la carpeta "/data" del projecte. Veiem que per passar la textura com a uniform indiquem el nom de l'uniform, la textura en si, i el canal de textura amb la funció *setUniformTexture*, que encapsula:

```
glActiveTexture(GL_TEXTURE0 + textureLocation);
glBindTexture(textureTarget,textureID);
setUniform1i(name,textureLocation);
glActiveTexture(GL_TEXTURE0);
```

Tal com hem vist a l'apartat de textures en GLSL 5.3.4.

7.3.5 Addons utilitzats

En aquesta secció veurem una pinzellada dels *addons* que s'han utilitzat en aquest projecte d'openFrameworks, explicant les funcions més importants. En veurem el detall en el Capítol 8.

ofxCubeMap

Aquest *addon* afegeix la possibilitat de treballar amb Cube Maps o *samplerCube*. Veiem com s'inicialitza un Cube Map:

```
// ofApp.h
// ...
ofxCubeMap cubemap; // Variable que actuara de Cube Map
// ...
// ofApp.cpp
// Carreguem les 6 imatges que formaran un cub
cubemap.loadImages("PATH/img_dreta.jpg",
                   "PATH/img_esquerra.jpg",
                   "PATH/img_dalt.jpg",
                   "PATH/img_baix.jpg",
                   "PATH/img_davant.jpg",
                   "PATH/img_darrera.jpg");
```

Si el que volem és utilitzar-lo com a textura, un cop carregat, hem d'habilitar-lo com a una textura a un dels canals utilitzant la següent funció:

```
ofxCubeMap::bindToTextureUnit(int pos)
```

pos és el número del canal. La funció activa el Cube Map com a textura en el canal *pos*:

```
glActiveTexture(GL_TEXTURE0+pos);
glEnable(GL_TEXTURE_CUBE_MAP);
glBindTexture(GL_TEXTURE_CUBE_MAP,textureObjectID);
```

Finalment, si el que volem es passar-ho com a *uniform* a un *shader* utilitzarem al funció:

```
shader.setUniform1i("myUniform",pos);
```

Just després d'haver-lo habilitat com a textura, el canal *pos* conté el Cube Map.

ofxAssimpModelLoader

Aquest *addon* afegeix la possibilitat de carregar i processar models 3D, en un format convenient i unificat. Permet carregar models del tipus: 3DS, OBJ, entre molts d'altres. Per carregar un model ho farem de la següent manera:

```
// ofApp.h
// ...
ofxAssimpModelLoader model;
// ...
// ofApp.cpp
model.loadModel("model_name.OBJ");
```

Entre d'altres funcions tenim:

```
void ofxAssimpModelLoader::setPosition(float x, float y, float z)
```

Per canviar la posició del model.

```
void ofxAssimpModelLoader::drawFaces()
```

Per pintar el model amb el tipus de *render* OF_MESH_FILL, que dibuixa els triangles com a geometria base d'OpenGL. És el tipus de "pintat" per defecte. Altres *renders* serien: OF_MESH_WIREFRAME, que pinta les línies, o bé OF_MESH_POINTS, que pinta els punts.

ofxUI

Aquest *addon* et permet afegir una interfície gràfica simplista al projecte d'openFrameworks. *ofxUI* està implementat a base de *widgets*, que permet col · locar-los lliurament, amb espais, tipus de font, guardar i/o carregar la configuració ... *ofxUI* es pot personalitzar molt fàcilment amb colors, fonts, mida dels *widgets*, distribució, etc.

Alguns dels *widgets* que incorpora *ofxUI* són: botons, etiquetes, *sliders*, llistes desplegables, *text input*, etc. Podem veure'ls a la següent imatge:

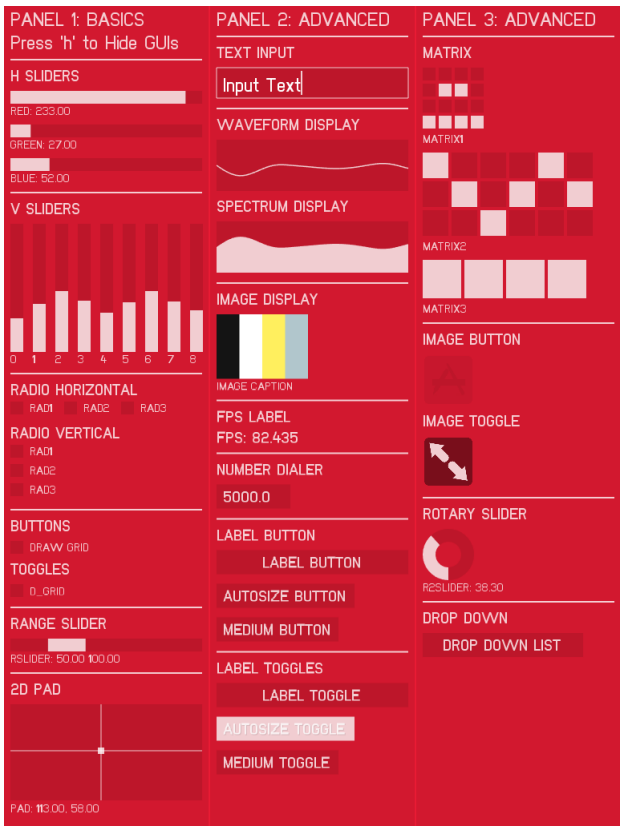


Figura 7.4: Interfície gràfica *ofxUI* amb alguns *widgets* d'exemple

Funcionament d'ofxUI

Un cop descarregat i instal · lat l'*addon*, el primer que es fa és incorporar aquestes dues funcions a l'aplicació:

```
void exit();
void guiEvent(ofxUIEventArgs &e);
```

La funció *exit* per defecte el que fa és destruir la UI un cop es tanca l'aplicació, però la funció important és la *guiEvent*. Aquesta funció funciona molt semblant a la funció *update*. Quan es detecta que hi ha hagut un *event* a algun dels *widgets* de la UI, la funció *guiEvent* és qui el tractarà.

Per declarar i inicialitzar la UI es fa de la següent manera:

```
// ofApp.h
// ...
ofxUICanvas *gui;
// ...
// ofApp.cpp (funcio setup)
gui = new ofxUICanvas();        //Creates a canvas at (0,0) using the default width
```

Ara ja només queda afegir tants *widgets* com vulguem i el *Listener* que "avisarà" la funció *guiEvent*. Per exemple (un cop creada):

```
  gui->addToggle("FULLSCREEN", false);
  gui->autoSizeToFitWidgets();
  ofAddListener(gui->newGUIEvent, this, &ofApp::guiEvent);
```

S'ha afegit una "palanca" que permetrà posar en pantalla completa la nostra aplicació. La funció *autoSizeToFitWidgets* col·loca els *widgets* de manera uniforme dins la UI. Finalment es crea el *Listener*. A continuació, si s'activa la "palanca" la funció *guiEvent* haurà de tractar l'*event*. Així doncs la funció quedaria de la següent manera:

```
void testApp::guiEvent(ofxUIEventArgs &e){
    if(e.getName() == "FULLSCREEN"){
        ofxUIToggle *toggle = e.getToggle();
        ofSetFullscreen(toggle->getValue());
    }
}
```

Es comprova qui ha causat l'*event*, a partir del nom que se li havia assignat, en aquest cas "FULLSCREEN" i es tracta. En aquest cas, s'agafa el valor de la "palanca" [*true*, *false*] i es crida la funció d'openFrameworks *ofSetFullscreen* que et permet posar en pantalla completa la finestra de l'aplicació.

7.3.6 Motius de la selecció

Vam estar investigant varis *frameworks* per intentar encastar-hi els *shaders* de 'Shadertoy': 'FreeFrame', 'Cinder', 'VVVV' i finalment openFrameworks.

La idea inicial era utilitzar 'FreeFrame' pels *shaders* i després fer la portabilitat a openFrameworks per jugar amb els models 3D. 'Cinder' i 'VVVV' van ser els dos primers descartats després de la reunió amb l'expert. 'FreeFrame' també vam haver de descartar-lo perquè estava desactualitzat des de feia molts anys i no tenia documentació, ni fòrum de debat entre usuaris. Vam trobar que openFrameworks era molt intuïtiu, contínuament actualitzat, amb molt bona documentació, usuaris implicats en els debats del fòrum, possibilitat de treballar amb extensions fetes pels usuaris (*addons*), encarat principalment per treballar amb OpenGL. Ho tenia tot per poder treballar bé amb *shaders*, d'una manera intuïtiva, eficaç i, el més important, educativa, gràcies a la bona documentació, tutorials i exemples.

7.4 C++

C++ és un llenguatge de programació que va ser creat, com el seu predecessor C, als laboratoris Bell de AT&T. El seu autor principal va ser Bjarne Stroustrup. L'any 1980 es van afegir noves característiques al llenguatge C, entre les principals la integració de les classes, idea que va ser presa de Simula67 (per molts considerat el primer llenguatge orientat a objectes). A partir d'aquí va anar evolucionant, fins que a l'any 1985 va ser consolidat com un llenguatge orientat a objectes i anomenat C++.

Actualment existeix un estàndard (ISO C++) al qual s'han adherit la majoria dels fabricants de compiladors més moderns. Existeixen també alguns intèrprets, com ara ROOT.

Una particularitat del C++ és la possibilitat de redefinir els operadors (sobrecàrrega d'operadors), i de poder crear nous tipus que es comportin com tipus fonamentals.

El nom C++ va ser proposat per Rick Mascitti l'any 1983, quan el llenguatge va ser utilitzat per primera vegada fora d'un laboratori científic. Abans s'havia fet servir el nom "C amb classes". L'expressió "C++" significa "increment de C" i es refereix al fet que C++ és una extensió de C.

C++ té els següents tipus fonamentals:

- Caràcters: *char* (també és un enter), *wchar_t*
- Enters: *short int*, *int*, *long int*, *long long int*
- Números en coma flotant: *float*, *double*, *long double*
- Booleans: *bool* (cert o fals)
- Buit: *void*

7.4.1 Motius de la selecció

openFrameworks està implementat en C++, i com és lògic s'utilitza en IDEs que utilitzen C++ com a llenguatge, per tant s'ha escollit el C++ com a llenguatge de programació per satisfer aquest requisit.

7.5 Python

Python és un llenguatge de programació creat per Guido van Rossum l'any 1990. Es compara habitualment amb TCL, Perl, Scheme, Java i Ruby. En l'actualitat Python es desenvolupa com un projecte de codi obert, administrat per la 'Python Software Foundation'. L'última versió estable del llenguatge és la 3.4.3.

Python és considerat com la "oposició lleial" a Perl, llenguatge amb el qual manté una rivalitat amistosa. Els usuaris de Python el consideren molt més net i elegant per a programar. Python permet dividir el programa en mòduls reutilitzables des d'altres programes Python. També hi ha mòduls inclosos que proporcionen E/S de fitxers, crides al sistema, *sockets* i fins interfícies gràfiques com: wxPython, GTK, Qt, entre d'altres.

Python és un llenguatge interpretat, és a dir, estalvia un temps considerable en el desenvolupament del programa, perquè no és necessari compilar ni enllaçar. L'intèrpret es pot utilitzar de manera interactiva, el qual facilita experimentar amb característiques del llenguatge, escriure programes d'un sol ús o provar funcions durant el desenvolupament del programa.

El principal objectiu que persegueix Python és la facilitat, tant de lectura, com de disseny.

7.5.1 Motius de la selecció

Per tal de poder descarregar els shaders de 'Shadertoy' necessitàvem un llenguatge que pogués "parsejar" la pàgina web. L'objectiu era poder obtenir el *fragment shader* de l'editor de text i construir els dos arxius *.vert* i *.frag*, tal com fa el motor de 'Shadertoy'. Això comportava que hauríem de fer connexions HTTP, actuar com a navegador per obtenir els continguts dels fitxers *javascript* un cop carregada la pàgina web, "parsejar" codi HTML, escriure a fitxers, entre d'altres funcionalitats.

Python permet fer tot això de manera molt intuïtiva i elegant. Altres llenguatges també ens haguessin servir, el C++ mateix, o fins i tot SCALA, però l'optimització del Python ha fet decantar la balança.

7.6 GanttProject

GanttProject és una aplicació que permet fer la organització de projectes. Està implementat en Java i és multiplataforma (Windows, Linux i OSX), sota la GPL-license. Permet organitzar el projecte en tasques i sub-tasques, indicant el període de temps que es trigarà en completar-les.

7.6.1 Motius de la selecció

S'ha escollit aquest programari per organitzar la planificació del treball a partir de tasques i subtasques, durant els mesos de recerca i desenvolupament.

7.7 StarUML

StarUML és un projecte de codi obert per al desenvolupament flexible i ràpid de diagrames UML/MDA per a Windows. L'objectiu d'aquesta plataforma és construir una eina de modelatge de *software* com a alternativa al *software* comercial d'UML tals com: Visual Paradigm, Rational Rose, Together, etc.

7.7.1 Motius de la selecció

El motiu de la selecció d'aquest programari és que permet desenvolupar tots els diagrames necessaris pel projecte, tals com: diagrames de classe, de cas d'ús, d'activitat, etc.

Capítol 8

Anàlisi i disseny del sistema

En aquest capítol veurem de manera detallada com s'ha analitzat i dissenyat el sistema partint de la problemàtica i seguint la metodologia exposades. Per a fer-ho explicarem el diagrama de cas d'ús general amb les fitxes corresponents i el diagrama de classes del sistema.

L'usuari de l'aplicació ha de poder descarregar *shaders* de 'Shadertoy' per poder executar-los al disc local, o bé "mapejar-los" en un model 3D com a simulació de *mapping*. Partint d'aquest objectiu s'han tingut en compte varies funcionalitats.

8.1 Diagrama i fitxes de cas d'ús

Per a desenvolupar l'aplicació s'han tingut en compte nombrosos factors amb què l'usuari podrà interactuar, alguns dels quals, com veurem, també requeriran desenvolupar-ne unes dependències per aconseguir el resultat.

Així doncs, el diagrama de cas d'ús de l'aplicació és el següent:

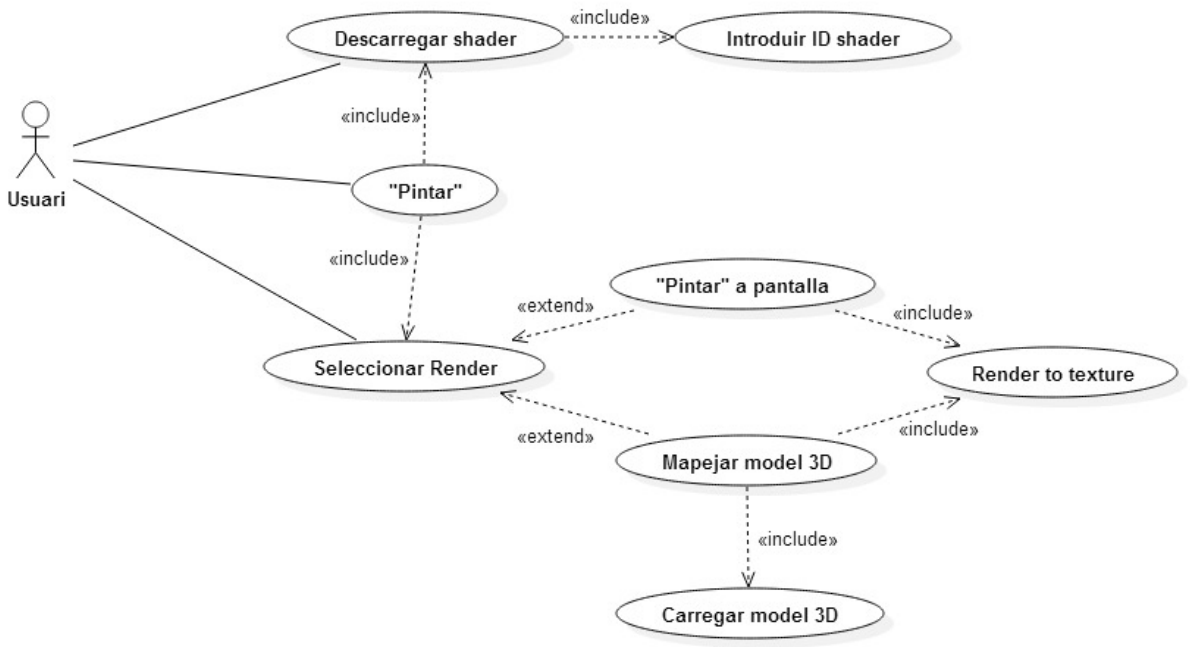


Figura 8.1: Diagrama de cas d'ús dels requeriments principals del sistema

L'aplicació ha de permetre a l'usuari poder descarregar un *shader* de 'Shadertoy' a partir de l'identificador. També ha de permetre seleccionar el tipus de *render* que es vol utilitzar per "pintar-lo": a pantalla o "mapejat" en un model 3D. En tots dos casos caldrà fer un *render to texture*, però a més, en el cas del model 3D, també caldrà poder importar-lo a l'aplicació.

Així doncs, les fitxes de cas d'us d'aquest requeriments són:

Fitxa de cas d'ús	**Descarregar shader**
Descripció	L'usuari vol descarregar un *shader* de 'Shadertoy'
Actor	Usuari
Pre-condició	L'usuari coneix l'identificador del *shader*
Flux principal	1. Introduir ID *shader* 2. "Parsejar" web 'Shadertoy' 3. Construir fitxers 3.1. Construir fitxer de textures 3.2. Construir *vertex shader* (*.vert*) 3.3. Construir *fragment shader* (*.frag*) 4. Escriure fitxers
Flux alternatiu	Si el *shader* ja l'hem descarregat un cop, la segona vegada que es vulgui utilitzar no es tornaran a crear els arxius de nou.
Post-condició	S'han descarregar el *vertex* i *fragment shader* i s'han guardat en els respectius arxius *.vert* i *.frag* en el disc local

Cal notar que el fitxer del punt 3.1 "Construir fitxer de textures" contindrà quines textures, el tipus, i en quin canal es troben emmagatzemades a 'Shadertoy'. Aquest fitxer serà de gran utilitat a l'hora de construir els *shaders* dins d'openFrameworks, ja que llegint aquest fitxer podrem saber quines i on s'utilitzen les textures, sense que l'usuari ho hagi de conèixer.

Fitxa de cas d'ús	**Introduir ID shader**
Descripció	L'usuari introdueix un ID d'un *shader* per descarregar
Actor	Usuari
Pre-condició	–
Flux principal	1. Escriure ID en el camp corresponent 2. Si ID correcte 2.1. Es pot descarregar el *shader* 3. Altrament 3.1. Error: "ID del *shader* és incorrecte"
Flux alternatiu	–
Post-condició	Si l'ID del *shader* és correcte es pot descarregar, altrament s'ha mostrat que hi ha hagut un error

Fitxa de cas d'ús	**"Pintar"**
Descripció	L'usuari vol "pintar" un *shader* de 'Shadertoy'
Actor	Usuari
Pre-condició	–
Flux principal	1. Descarregar *shader* 2. Seleccionar render (pantalla o model 3D) 3. "Pintar" segons render
Flux alternatiu	–
Post-condició	S'ha pintat el *shader* descarregat amb el render seleccionat

Fitxa de cas d'ús	**Seleccionar render**
Descripció	Selecció del tipus de render per "pintar"
Actor	Usuari
Pre-condició	–
Flux principal	1. Seleccionar tipus de render (desplegable) 1.1. Seleccionar "pintar" a pantalla, o bé 1.2. Seleccionar "mapejar" en un model 3D 1.2.1. Entrar nom del model en el camp corresponent
Flux alternatiu	No seleccionar res i per defecte es "pinta" a pantalla
Post-condició	S'ha seleccionat el tipus de render amb el qual es "pintarà"

Fitxa de cas d'ús	**"Pintar" a pantalla**
Descripció	Es "pinta" el *shader* de 'Shadertoy' a pantalla
Actor	Sistema
Pre-condició	L'usuari ha seleccionat "pintar" a pantalla com a tipus de render i el *shader* s'ha descarregat correctament
Flux principal	1. Passar *uniforms* al *shader* 2. Render to texture del *shader* 3. "Pintar" la nova textura en un rectangle de la mida de la finestra
Flux alternatiu	–
Post-condició	S'ha "pintat" el *shader* a pantalla

Fitxa de cas d'ús	**Mapejar model 3D**
Descripció	Es "mapeja" el *shader* de 'Shadertoy' en un model 3D
Actor	Sistema
Pre-condició	L'usuari ha seleccionat "mapejar" en un model 3D com a tipus de render i el *shader* s'ha descarregat correctament
Flux principal	1. Comprovar nom entrat del model 1.1. Si existeix el model 1.1.1. Continuar 1.2. Altrament 1.2.1. Error: "El model indicat no existeix" 2. Carregar model 3D 3. Passar *uniforms* al *shader* 4. Render to texture del *shader* 5. "Mapejar" la nova textura en el model 3D carregat 5.1. Moure model segons ratolí i tecles WASD
Flux alternatiu	–
Post-condició	S'ha "mapejat" el *shader* en el model 3D carregat

Fitxa de cas d'ús	**Carregar model 3D**
Descripció	Càrrega del model 3D indicat per l'usuari, per posterior "mapejat"
Actor	Sistema
Pre-condició	El model indicat per l'usuari existeix
Flux principal	1. Carregar un model a partir d'un arxiu OBJ 2. Col · locar el model ajustat a la finestra
Flux alternatiu	–
Post-condició	S'ha carregat el model indicat per l'usuari

Fitxa de cas d'ús	**Render to texture**
Descripció	*Render to texture* del *shader* escollit per l'usuari
Actor	Sistema
Pre-condició	El *shader* s'ha carregat correctament
Flux principal	1. Inicialització FBO 2. "Pintar" el *shader* dins el FBO
Flux alternatiu	–
Post-condició	S'ha pintat el *shader* indicat en un *frame buffer object* (*render to texture*)

8.2 Diagrama de classes

En aquesta secció veurem com s'ha dissenyat el diagrama de classes (Figura 8.2), tenint en compte l'estructura d'openFrameworks i els *addons* utilitzats.

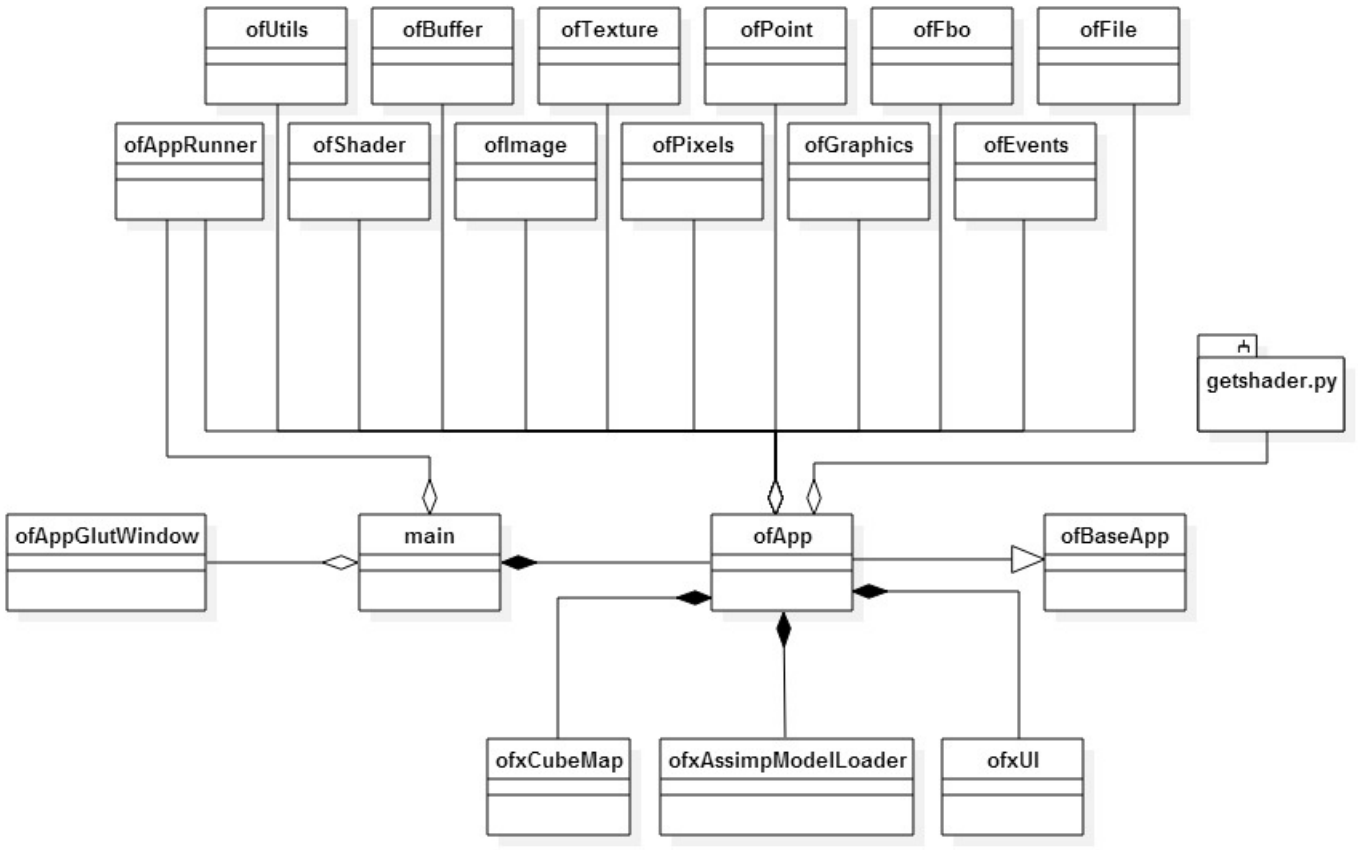


Figura 8.2: Diagrama de classes de l'aplicació

Com s'ha vist en la Secció 7.3.3, l'entorn openFrameworks ve donat per una classe principal *ofApp*, la qual hereta d'una classe *ofBaseApp*. A més, l'entorn també incorpora una classe *main* que crea la finestra on s'executarà l'aplicació amb *ofAppGlutWindow* i una classe *ofAppRunner* que executa l'aplicació. Com es pot veure la classe *ofApp*, que implementa l'aplicació, utilitza moltes altres classes, totes amb prefix *of*. Aquestes classes són les utilitzades del *framework*, això vol dir que només s'han utilitzat les funcions que han estat necessàries d'aquestes classes.

També, com es pot veure, la classe *ofApp* incorpora unes altres tres classes, que són *addons* d'openFrameworks. Aquestes classes són: *ofxCubeMap* per executar *shaders* amb *cubeMaps*, *ofxAssimpModelLoader* per poder carregar models 3D i *ofxUI* per a la interfície gràfica. Veiem també que la classe *ofApp* té relació amb un subsistema *getshader.py*, que és l'*script* en Python que descarrega els *shaders* de 'Shadertoy'.

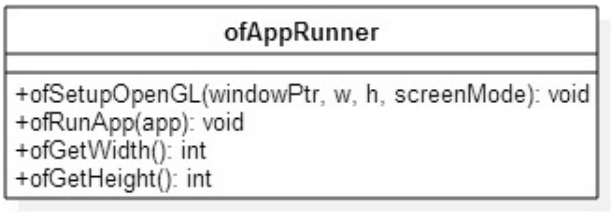
És evident que tant les classes d'openFrameworks com els *addons* utilitzats tenen relació amb altres classes o, fins i tot, *addons*, però com que per aquesta aplicació no s'han utilitzat, no les incorporem en el diagrama de *classes*.

A continuació veurem individualment per a cada classe, els mètodes i atributs necessaris. En el cas de les classes d'openFrameworks i els *addons* s'indicaran els mètodes utilitzats i per a què serveixen. Els mètodes de l'aplicació es desenvoluparan en pseudocodi.

8.3 Les classes

En aquesta secció veurem en més detall les classes d'openFrameworks que s'han utilitzat, explicant els mètodes que s'han utilitzat i quina funcionalitat tenen. També farem el mateix per part dels tres *addons* utilitzats, i finalment les classes a implementar dels mètodes per a l'aplicació.

8.3.1 ofAppRunner



La classe *ofAppRunner* és el motor de les aplicacions desenvolupades en openFrameworks. És qui s'encarrega de cridar les funcions principals de l'*ofApp*. Veiem-ne quins mètodes s'han utilitzat:

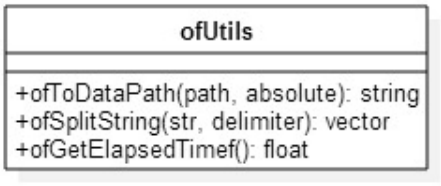
- **ofSetupOpenGL(windowPtr,w,h,screenMode)**: Configura l'aspecte i el mode de la finestra. Aquest funció només ha de ser cridada a la funció *main* del *main.cpp*. *w* i *h* són l'amplada i l'alçada de la finestra, mentre que *screenMode* és el mode de la pantalla: OF_WINDOW per pantalla normal o OF_FULLSCREEN per pantalla completa.

- **ofRunApp(app)**: És la funció principal de l'entorn openFrameworks. Inicia el cicle OpenGL de l'aplicació. Només és cridada a la funció *main* del *main.cpp*, just després de configurar la finestra amb *ofSetupOpenGL*.

- **ofGetWidth()**: Retorna l'amplada de la finestra de l'aplicació.

- **ofGetHeight()**: Retorna l'alçada de la finestra de l'aplicació.

8.3.2 ofUtils



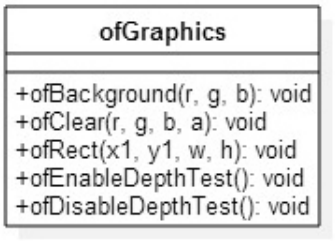
Aquesta classe proporciona algunes funcions que s'utilitzen habitualment en programació, com pot ser per la manipulació d'*strings*, conversió de format de números, funció *random*, obtenir el temps, etc. Veiem-ne quins mètodes s'han utilitzat:

– **ofToDataPath(path,absolute)**: Converteix la ubicació donada a una ubicació relativa a la carpeta */data* del projecte. *path* és la ubicació donada i *absolute* un booleà que indica si la ubicació és absoluta.

– **ofSplitString(str,delimiter)**: Divideix un *string* donat en parts separades per un delimitador. *str* és l'*string* donat i *delimiter* el delimitador. Per exemple: *str* = "Hello,world" i delimiter = ',' obtindríem un vector de dos *strings* [Hello,world].

– **ofGetElapsedTimef()**: Retorna el temps transcorregut en segons des de l'inici de l'aplicació, com a *float*.

8.3.3 ofGraphics



La classe *ofGraphics* proporciona les funcions bàsiques per *render*, com ara: "pintar" primitives, manipulació de primitives, manipulació de matrius, utilització d'*alpha blending*[1], *antialiasing*[2], manipulació del *viewport*, etc.

– **ofBackground(r,g,b)**: Estableix el color de fons de l'aplicació. L'*input* és el color en format RGB, cada paràmetre pot valer de 0 a 255.

– **ofClear(r,g,b,a)**: Esborra els bits de color i profunditat del *renderer* actual i el reemplaça per un color RGBA.

– **ofRect(x1,y1,w,h)**: Dibuixa un rectangle des del punt (x1,y1) amb amplada *w* i alçada *h*. Per exemple: si *x1*=0, *y1*=0, *w*=amplada de la finestra i *h*=alçada de la finestra, dibuixarà un rectangle des de l'origen de coordenades (extrem superior esquerra) amb dimension de la finestra, és a dir, ocupara tota la finestra.

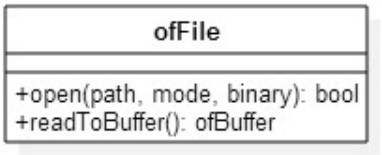
– **ofEnableDepthTest()**: Activa el test de profunditat, així el renderitzat es fa d'acord amb el *z-depth* en comptes de l'ordre de pintat. És a dir, quan es renderitza l'escena es té en compte en quina profunditat es troben les figures i no en quin ordre s'han pintat, ja que d'aquesta manera, sinó, podríem veure més aprop una figura que es troba més lluny.

– **ofDisableDepthTest()**: Desactiva el test de profunditat, així el renderitzat es fa d'acord amb l'ordre de pintat en comptes del *z-depth*

[1] *Alpha Blending* és la tècnica de combinar el canal *alpha* amb altres capes en una imatge, per tal de simular transparència. Per exemple: una figura amb colors RGBA, la 'A' indicarà en nivell de transparència del color RGB.

[2] L'*antialiasing* és la tècnica de minimitzar la distorsió (*aliasing*) quan es representa una imatge de baixa resolució en resolució més alta.

8.3.4 ofFile

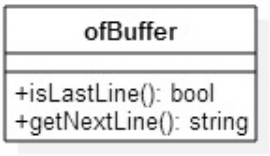


La classe *ofFile* agrupa les funcionalitats per obrir, llegir, escriure, i modificar fitxers emmagatzemats en el disc local. Les funcions utilitzades són les següents:

– **open(path,mode,binary)**: Obre el fitxer que es troba a *path* amb el *mode* que pot ser: *Reference*, *ReadOnly*, *WriteOnly*, *ReadWrite*, *Append* i indicant si el fitxer és binari o no.

– **readToBuffer()**: Escriu el contingut del fitxer en un *ofBuffer* i el retorna.

8.3.5 ofBuffer

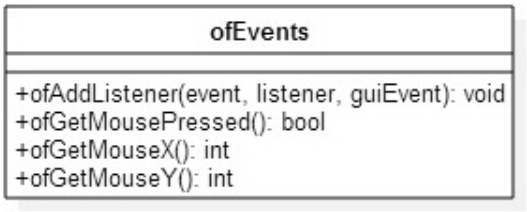


La classe *ofBuffer* proporciona mètodes senzilles per a la lectura i escriptura de fitxers. És un bon complement a la classe *ofFile*. A més, incorpora diferents maneres per convertir el contingut d'un fitxer en *strings*. Les funcions utilitzades són les següents:

– **isLastLine()**: Retorna un booleà indicant si el text actual és la última línia del fitxer.

– **getNextLine()**: Retorna la línia de text abans del següent salt de línia.

8.3.6 ofEvents



La classe *ofEvents* és qui s'encarrega de controlar els *events* que poden haver-hi durant l'execució de l'aplicació, com pot ser: *event* del ratolí o teclat, *events* de la interfície gràfica, missatges entre classes, *events* tàctils (en el cas d'utilitzar un *pad*), etc. Les funcions utilitzades són les següents:

– **ofAddListener(event,listener,guiEvent)**: Permet afegir un *listener* que "escoltarà" un tipus d'esdeveniment donat, que pugui haver-hi durant l'execució de l'aplicació. *event* és

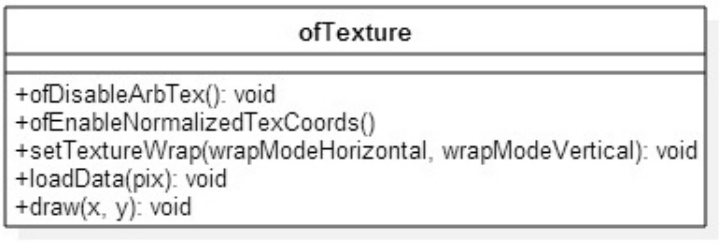
el tipus d'esdeveniment que s'escoltarà, *listener* és qui s'encarregarà d'escoltar i *guiEvent* la funció on es tractarà l'esdeveniment. En aquest cas es diu *guiEvent*, perquè s'utilitzarà el mètode per escoltar i tractar els *events* de la GUI.

– **ofGetMousePressed()**: Retorna un booleà indicant si s'està prement algun botó del ratolí.

– **ofGetMouseX()**: Retorna la posició del ratolí sobre l'eix de les X, del sistema de coordenades de la pantalla.

– **ofGetMouseY()**: Retorna la posició del ratolí sobre l'eix de les Y, del sistema de coordenades de la pantalla.

8.3.7 ofTexture



La classe *ofTexture* encapsula l'API de les textures d'OpenGL. Permet crear textures que es podran utilitzar per omplir objectes pintats. A diferència de les textures d'OpenGL, no necessàriament han de tenir dimensions potència de 2. Veiem quins mètodes s'han utilitzat:

– **ofDisableArbTex()**: Desactiva la necessitat de passar les coordenades de textura com a "pixels", és a dir, es passen les coordenades normalitzades entre 0-1. Per defecte openFrameworks treballa amb textures ARB, les quals en GLSL s'utilitzen *sampler2DRect*, en comptes de *sampler2D*, que permet utilitzar textures amb mides que no siguin potència de 2.

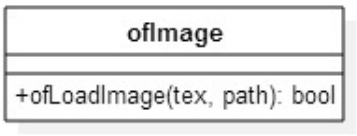
– **ofEnableNormalizedTexCoords()**: Activa la utilització de coordenades de textura normalitzades (0-1). Les coordenades normalitzades (0-1) són les coordenades per defecte en l'entorn GL. Permeten utilitzar el contingut de la textura sense haver de preocupar-se per les dimensions.

– **setTextureWrap(wrapModeHorizontal,wrapModeVertical)**: Estableix com la textura s'adapta al voltant de les vores dels vèrtexs on s'està dibuixant. Per defecte s'estiren les vores de les textura fins emplenar la superfície on s'està "pintant" (GL_CLAMP_TO _EDGE), en canvi, si el que es vol es repetir la textura fins emplenar la superfície, es fa amb GL_REPEAT.

– **loadData(pix)**: Carrega els píxels d'una instància *ofPixels* com a contingut de la textura.

– **draw(x,y)**: "Pinta" la textura amb origen al punt (x,y) donat, utilitzant les mides d'amplada i alçada reals, és a dir, les dimensions normals que té la textura.

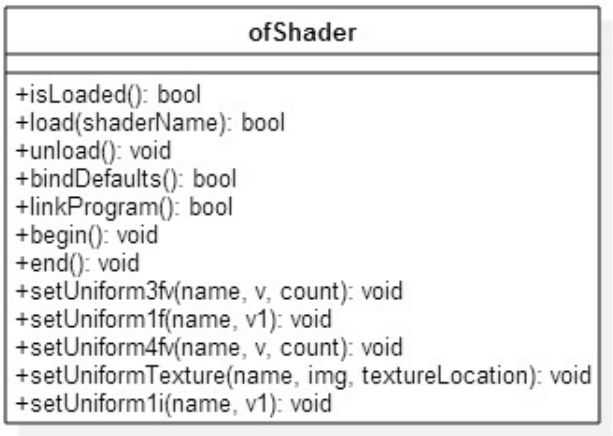
8.3.8 ofImage



La classe *ofImage* és útil per carregar, guardar i "pintar" imatges en openFrameworks. Permet tant "pintar" imatges a pantalla, com manipular-ne els seus píxels. Permet carregar imatges del disc local, manipular-ne els píxels, i crear-ne una textura d'OpenGL que es pot visualitzar i manipular a la targeta gràfica. Els mètodes utilitzats són els següents:

– **ofLoadImage(tex,path)**: Permet carregar la imatge que es troba a *path* com a contingut de la textura *tex*, per posteriorment tractar-la com a textura d'OpenGL.

8.3.9 ofShader



La classe *ofShader* permet la utilització de *shaders* (GLSL), tal com s'ha vist en el Capítol 5.3. Aquesta classe encapsula els mètodes més importants per a la interacció amb *shaders* (pas d'*uniforms* i *attributes*), inicialitzar i enllaçar el *vertex* i *fragment shader*, etc. Els mètodes útilitzats són els següents:

– **isLoaded()**: Retorna un booleà que indica si el *shader* s'ha carregat correctament.

– **load(shaderName)**: S'assumeix que tant el *vertex* com el *fragment shader* tenen el mateix nom, per exemple: *shader.vert* i *shader.frag*, i els carrega utilitzant només el nom del *shader* (*shaderName*).

– **unload()**: Descarrega el *shader*, que significa que no estarà actiu més temps a la targeta gràfica.

– **bindDefaults()**: Lliga els *attributes* per defecte que sol tenir un *shader*: *position*, *color*, *normal* i *texcoord* en una posició dins les 16 possibles (límit del *hardware*), tenint en compte la mida. Per exemple, una *mat4* ocuparà 4 posicions, per tant no lligarà aquesta *mat4* a l'índex 0 i un *float* a l'índex 1, ja que la *mat4* ocuparà els index 0 a 3.

– **linkProgram()**: Enllaça el programa *GLuint* (el programa que s'encarregarà del funcionament dels *shaders*) amb els *shaders* compilats.

– **begin()**: Després de cridar aquesta funció, tot el que es "pinti", vèrtex i textures, a l'aplicació tindrà els efectes del *shader* que se'ls aplica.

– **end()**: Després de cridar aquesta funció, tot el que es "pinti", vèrtex i textures, a l'aplicació, no tindrà cap efecte del *shader* que se'ls aplica.

– **setUniform3fv(name,v,count)**: Permet passar un vector de 3 *floats v* com a *uniform*, de nom *name*, al *shader*. *count* indica el nombre d'elements que han de ser modificats.

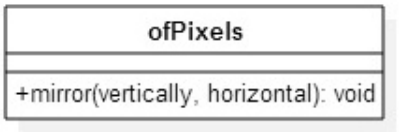
– **setUniform1f(name,v1)**: Permet passar un *float* com a *uniform*, de nom *name*, al *shader*. El valor del *float* és *v1*.

– **setUniform4fv(name,v,count)**: Permet passar un vector de 4 *floats v* com a *uniform*, de nom *name*, al *shader*. *count* indica el nombre d'elements que han de ser modificats.

– **setUniformTexture(name,img,textureLocation)**: Permet passar textures com a *uniforms* al *shader*. *name* és el nom de l'*uniform*, *img* la textura i *textureLocation* el canal de l'*uniform*.

– **setUniform1i(name,v1)**: Permet passar un enter com a *uniform*, de nom *name*, al *shader*. El valor de l'enter és *v1*.

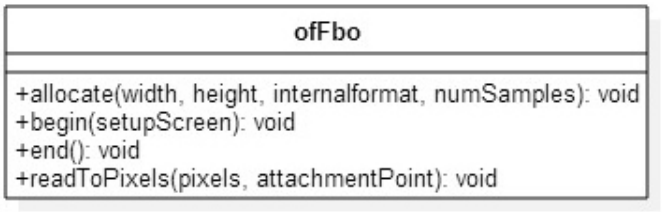
8.3.10 ofPixels



ofPixels és un objecte per treballar amb blocs de píxels, aquests píxels poden ser copiats d'una imatge que s'hagi carregat, quelcom que s'hagi "pintat" utilitzant *ofGraphics*, etc. Es pot crear una imatge a partir de píxels utilitzant *ofPixels*. La funció que s'ha utilitzat és la següent:

– **mirror(vertically,horizontal)**: Donada una representació de píxels, la funció *mirror* permet voltejar, com si fos un mirall, aquests píxels, tant de manera vertical com horitzontal. Tant *vertically* com *horizontal* són booleans.

8.3.11 ofFbo



L'objecte *ofFbo* encapsula els mètodes per tractar *Framebuffer Objects* d'OpenGL, alguns dels quals s'han vist a la Secció 5.3.5. Un *ofFbo* és un contenidor per textures, on normalment en OpenGL s'utilitzen per renderitzar-hi. S'hi poden "pintar" textures, objectes 2D i 3D, amb la diferència que és un objecte emmagatzemat a la targeta gràfica que representa un pas de renderitzat. Els mètodes utilitzats són els següents:

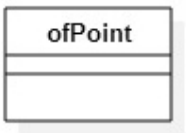
- **allocate(width,height,internalformat,numSamples)**: Aquesta funció permet establir l'amplada, alçada, i el tipus GL del FBO (per exemple si té dades *alpha* o no, GL_RGBA), a més del número de mostres per MSAA[3].

- **begin(setupScreen)**: Qualsevol "pintat" que es realitzi després de cridar la funció *begin*, es "pintarà" dins el FBO en comptes de a pantalla. *setupScreen* és un booleà que indica si es vol configurar la perspectiva de la pantalla (amplada, alçada, *fov*, *zNear*, *zFar*, etc.).

- **end()**: Qualsevol "pintat" que es realitzi després de cridar la funció *end*, es deixarà de "pintar" dins el FBO i es tornarà a pantalla.

- **readToPixels(pixels,attachmentPoint)**: Permet obtenir els píxels d'un *ofFbo* i emmagatzemar-los en una instància *ofPixels*. El paràmetre *attachmentPoint* permet indicar quina de les textures unides al FBO es vol agafar.

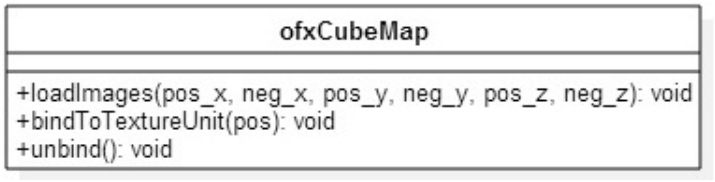
8.3.12 ofPoint



Aquesta classe és un *typedef* de ofVec3f, la qual permet emmagatzemar un vector tridimensional. En aquesta aplicació només s'ha utilitzat per emmagatzemar les coordenades del cursor, per a la funció *mouseDragged*, i poder rotar i desplaçar verticalment el model 3D utilitzant el ratolí.

[3]El *Multisample anti-aliasing* (MSAA) és un tipus d'*anti-aliasing*. Pren mostres d'ombrejat, textura i color en els píxels frontera dels elements, generant un promig entre totes les mostres. El MSAA només afecta a les vores i no a la part interna de l'objecte.

8.3.13 ofxCubeMap



ofxCubeMap és un *addon* per openFrameworks que incorpora les funcionalitats dels *cubeMaps* d'OpenGL. Un *cubeMap* és una agrupació de 6 imatges en 2D, les quals formen un cub. És molt útil alhora de crear escenografies de manera fàcil.

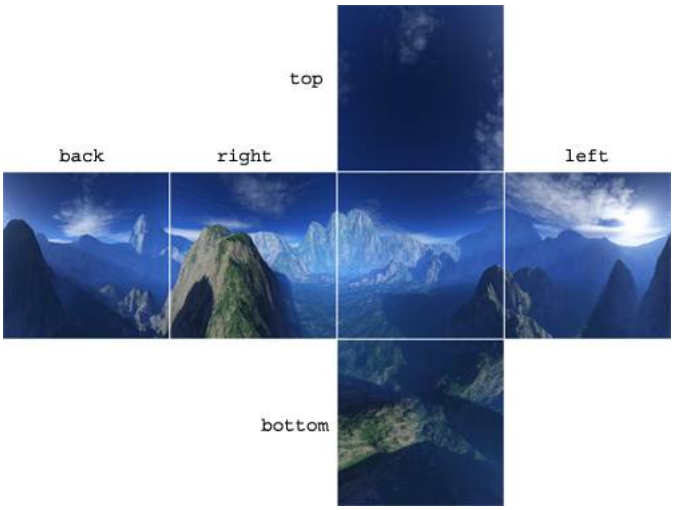


Figura 8.3: Representació d'un *cubeMap* format a partir de 6 imatges.

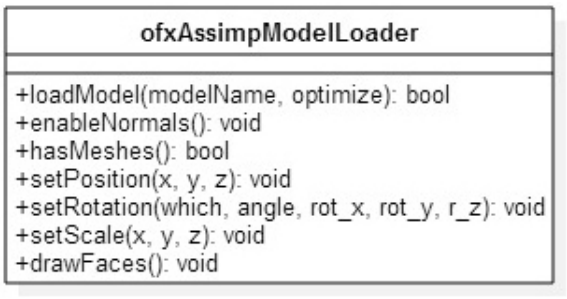
Les funcions utilitzades són les següents:

- **loadImages(pos_x,neg_x,pos_y,neg_y,pos_z,neg_z)**: Permet carregar les 6 imatges que actuaran com a cares del cub. Es carreguen en ordre dels eixos {X,Y,Z} i coordenades {positives, negatives}.

- **bindToTexture(pos)**: Carrega el *cubeMap* com una textura al canal *pos*. El procediment per carregar és el mateix que s'ha vist a la Secció 5.3.4, però en comptes d'enllaçar GL_TEXTURE2D s'enllaça GL_TEXTURE_CUBE_MAP.

- **unbind()**: Desenllaça el *cubeMap* del canal on es trobi.

8.3.14 ofxAssimpModelLoader



ofxAssimpModelLoader és un *addon* per openFrameworks que permet carregar models 3D, en un format convenient, a memòria, i processar-los. Aquest *addon*, sobretot, encapsula la llibreria *Open Asset Import Library* (*assimp*). Alguns dels formats suportats són: OBJ, 3DS, ASE, STL, etc. Les funcions utilitzades són les següents:

- **loadModel(modelName,optimize)**: Permet carregar un model 3D, de nom *modelName*, i indicar-ne si se'n vol optimitzar els vèrtexs, *meshes*[4], la memòria cau, etc. Per defecte *optimize=false*.

- **enableNormals()**: Activa la utilització de vectors *normal*[5] del model 3D.

- **hasMeshes()**: Retorna un booleà que indica si el model 3D té, o no, meshes.

- **setPosition(x,y,z)**: Configura la posició del model (*model matrix*) al punt {x,y,z} sobre el sistema de coordenades del model.

- **setRotation(which,angle,rot_x,rot_y,rot_z)**: Permet aplicar una rotació al model 3D. *which* indica la mesh que es rotarà, amb un *angle*. *rot_x*, *rot_y* i *rot_z* valdran 1 o 0 si es vol, o no, rotar sobre un o més eixos.

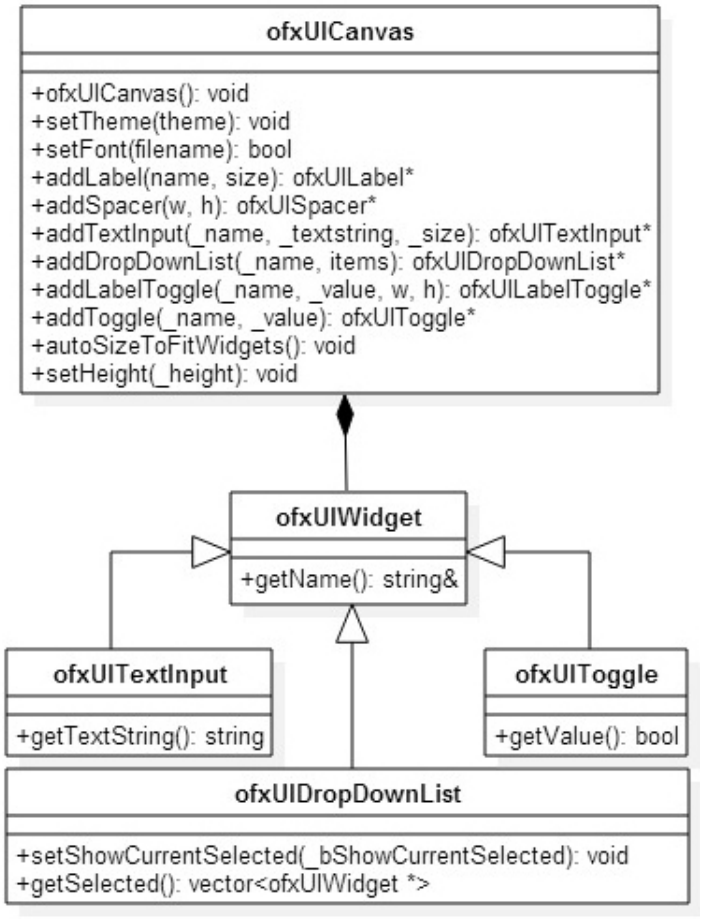
- **setScale(x,y,z)**: Permet escalar el model en funció dels paràmetres {x,y,z}. Per exemple: si es vol fer el model el doble d'alt: *setScale(1,2,1)*.

- **drawFaces()**: "Pinta" les cares del model a partir dels triangles que formen les *meshes* (OF_MESH_FILL).

[4]Una *mesh* (malla poligonal) és una superfície creada a partir de diverses primitives. Per exemple: una *mesh* de 2 triangles que formen la cara d'un cub.

[5]Un vector *normal* és un vector perpendicular a la superfície donada. En OpenGL es solen utilitzar pels càlculs d'il · luminació.

8.3.15 ofxUI



ofxUI és un *addon* per openFrameworks que permet crear interfícies d'usuari i GUIs. *ofxUI* treballa amb un disseny de *widgets*, espais, carrega de fonts, guardar i carregar configuracions, que pot ser fàcilment configurable tant a nivell de color, mides, distribució dels *widgets*, etc. Alguns dels *widgets* que s'hi poden trobar són: botons, etiquetes, palanques, entrades de text, 2D *pads*, *sliders*, etc.

Aquest *addon* incorpora diverses classes, en aquest cas les que s'han utilitzant són la *ofxUI-Canvas* per crear l'entorn de la GUI i la *ofxUIWidget* per accedir als *widgets* del *canvas*.

Les funcions utilitzades d'aquestes classes són:

- **ofxUICanvas()**: Constructor de la classe *ofxUICanvas*. Crea una instància d'un *canvas* per ser utilitzat de GUI.

- **setTheme(theme)**: Permet establir una combinació de colors (*tema*) predefinits dins l'*addon*. *theme* és un enter que indica el número del tema.

- **setFont(filename)**: Permet carregar un tipus de font a partir d'un arxiu de font (per exemple extensió *.ttf*), per ser utilitzat com a tipus de lletra de la GUI.

- **addLabel(name,size)**: Afegeix una etiqueta a la GUI. *name* és el nom de l'etiqueta i, per tant, el que s'escriurà a la GUI, mentre que *size* indica la mida de la font amb què

s'escriurà l'etiqueta.

– **addSpacer(w,h)**: Afegeix un espai d'amplada w i alçada h a la GUI, utilitzant el color del tema. Si $w = 0$, l'espai es transparent.

– **addTextInput(_name,_textstring,_size)**: Afegeix una entrada de text a la GUI. *_name* és l'identificador, *_textstring* el text inicial i *_size* la mida de la font.

– **addDropDownList(_name,items)**: Afegeix una llista de *strings* en forma de desplegable. *_name* és l'identificador i *items* un vector de *strings*.

– **addLabelToggle(_name,_value,w,h)**: Afegeix una palanca en forma de botó. *_name* és l'identificador, *_value* indica si està premut o no (*true* o *false*), w i h són l'amplada i l'alçada respectivament.

– **addToggle(_name,_value)**: Afegeix una palanca en forma de *checkbox*. *_name* és l'identificador i *_value* indica si està premuda o no (*true* o *false*).

– **autoSizeToFitWidgets()**: Un cop afegits tots els *widgets*, es crida aquesta funció perquè els col · loqui proporcionalment distribuïts dins el *canvas* de la GUI.

– **setHeight(_height)**: Estableix una alçada al *canvas* de la GUI.

– **getName()**: Donat un *widget*, en retorna l'identificador (*name*).

– **getTextString()**: Donat un *text input*, retorna el text que té emmagatzemat.

– **getValue()**: Donat un *toggle* (palanca), retorna si està activa o no (*true* o *false*).

– **setShowCurrentSelected(_bShowCurrentSelected)**: Donada una llista desplegable, aquesta funció permet que es mostri l'element seleccionat un cop plegada. *_bShowCurrentSelected* és un booleà.

– **getSelected()**: Retorna l'element seleccionat de la llista desplegable.

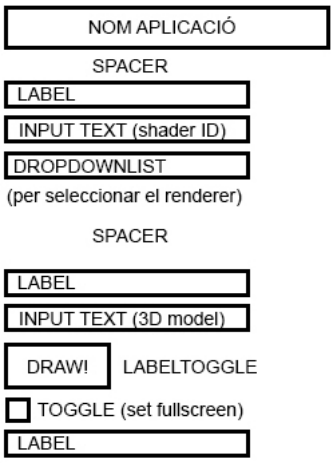
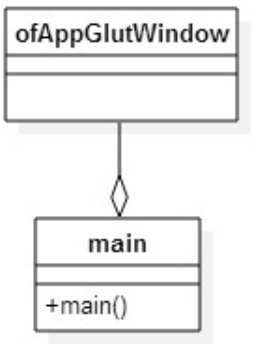


Figura 8.4: Esquema de com haurà de ser la GUI de l'aplicació

El disseny de la GUI contindrà, com a *widgets* més importants, dos *ofxUITextInput*, per indicar l'ID del *shader* de 'Shadertoy', i el nom del model 3D, si es vol "mapejar" sobre un model. També una *ofxUIDropDownList* per escollir el tipus de render (a pantalla o "mapejar" sobre un model 3D). Finalment, un *ofxUILabelToggle* per executar l'aplicació amb els paràmetres dels *widgets*.

8.3.16 main



La classe *main* és la classe principal d'una aplicació en openFrameworks, s'encarrega de crear, configurar l'aspecte i el mode de la finestra on s'executarà l'aplicació. A més, es comunica amb *ofAppRunner* per executar l'aplicació.

L'únic mètode que s'ha d'implementar en aquesta classe és el *main()*. Dins aquest mètode es crea la finestra i s'executa l'aplicació. Veiem com seria:

```
FUNCIO main() {
  // Declarem la finestra
  ofAppGlutWindow finestra;

  // Configurem la finestra
  configurarFinestra(finestra,width,height);

  // Executem l'aplicacio
  ofRunApp(app);

}
```

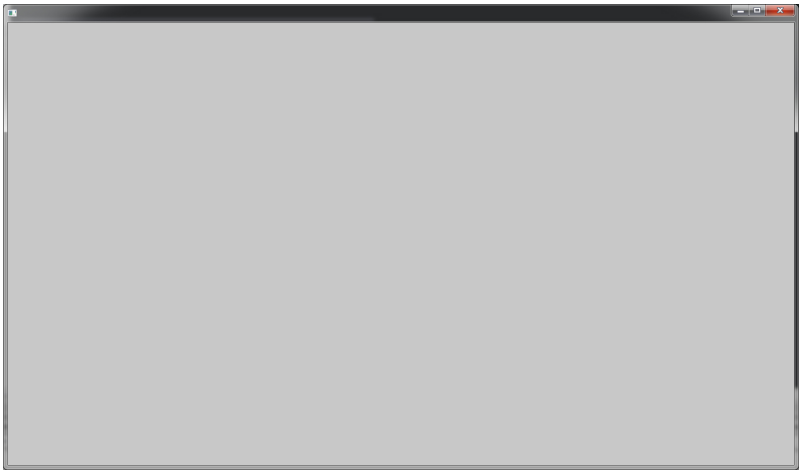
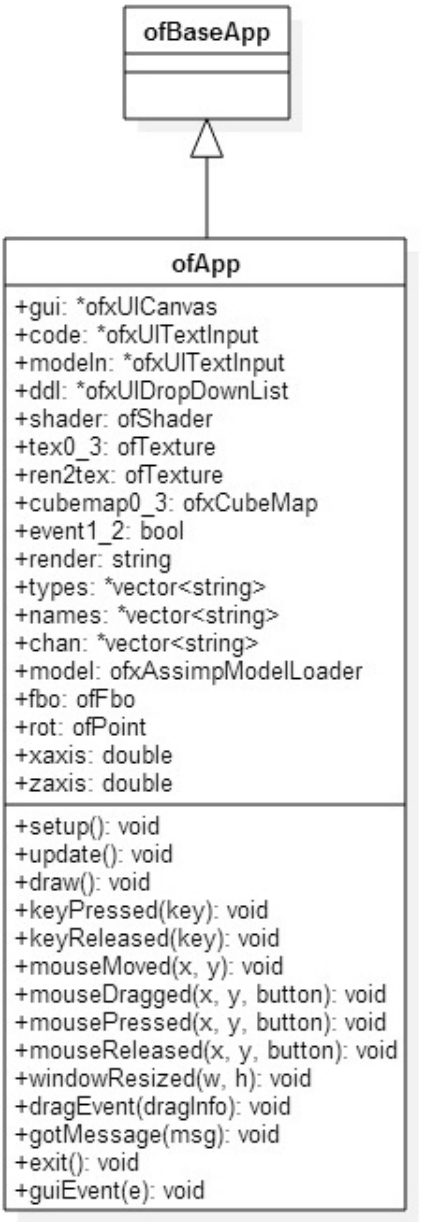


Figura 8.5: Finestra d'openFrameworks configurada a 1280x720 píxels

8.3.17 ofApp



La classe *ofApp* hereta de la classe *ofBaseApp*, la qual és el motor de l'aplicació en una aplicació d'openFrameworks. Quan es crea un nou projecte es crea una instància *ofApp* per sobreescriure els mètodes definits a *ofBaseApp*, com ara *setup*, *update* i *draw*. Els mètodes que trobem a *ofBaseApp* i que poden ser sobreescrits a *ofApp* són:

- **setup()**: Funció per configurar els valors inicials, tals com variables globals o la GUI de l'aplicació. És la primera funció que s'executa i només ho fa en el moment d'inicialitzar l'aplicació.

- **update()**: Funció que es crida repetidament. És cridada just abans de "pintar", *draw()*, i s'utilitza per actualitzar variables si, per exemple, es compleixen unes certes restriccions.

- **draw()**: Funció que es crida repetidament. És cridada just després de la funció *update()*. És on es renderitzen els resultats i es "pinten" els elements.

- **keyPressed(key)**: La funció és cridada quan es prem una tecla. *key* és la tecla premuda, per exemple *key* = 'a'. Dins la funció es tracta l'*event* en funció de la tecla premuda.

- **keyReleased(key)**: La funció és cridada quan es deixa de prémer una tecla. *key* és la tecla que es deixa de prémer, per exemple *key* = 'a'. Dins la funció es tracta l'*event* en funció de la tecla que es deixa de prémer.

- **mouseMoved(x,y)**: La funció és cridada quan es mou el ratolí. x i y són les coordenades en píxels de pantalla de la posició del ratolí.

- **mouseDragged(x,y,button)**: La funció és cridada quan el ratolí s'està movent amb un botó premut. x i y són les coordenades en píxels de pantalla de la posició del ratolí, i *button* indica quin dels botons del ratolí s'està prement.

- **mousePressed(x,y,button)**: La funció és cridada quan es prem un botó del ratolí. La funció rep les coordenades en píxels de pantalla de la posició del ratolí (x i y) i el botó que s'esta prement (dreta, centre, esquerra).

- **mouseReleased(x,y,button)**: La funció és cridada quan es deixa de prémer un botó del ratolí. La funció rep les coordenades en píxels de pantalla de la posició del ratolí (x i y) i el botó que s'ha deixat de prémer (dreta, centre, esquerra).

- **windowResized(w,h)**: La funció és cridada quan es canvia la mida de la pantalla. La funció rep la nova amplada w i alçada h.

- **dragEvent(dragInfo)**: La funció *dragEvent* controla el contingut que s'arrossega dins l'aplicació des de fora. *dragInfo* és un vector de *strings* que conté els noms dels arxius que s'estan arrossegant cap a l'aplicació.

- **gotMessage(msg)**: *msg* és un *event* que conté un *string*, que s'envia quan s'envia un altre *event*. S'utilitza per personalitzar el contingut d'un *event*, per obtenir més informació i ser tractada dins *gotMessage*.

Així doncs, els mètodes principals a implementar per a l'aplicació són: *setup*, *update* i *draw*, amb l'ajuda dels mètodes que facin falta per controlar els *events* de l'aplicació, que també s'hauran d'implementar.

A banda d'aquests mètodes, se'n poden afegir tants com facin falta. En el cas de l'*addon ofxUI*, s'han d'implementar un parell més de funcions, que són:

- **exit()**: La funcionalitat d'aquest mètode és la d'eliminar la UI un cop es tanca l'aplicació.

- **guiEvent(e)**: Funció que permetrà controlar els *events* que es produeixin a la UI durant l'execució de l'aplicació. e és l'*event* a controlar.

Els atributs que es necessitaran pel correcte funcionament de l'aplicació s'explicaran a la Secció 8.4, alhora que es planteja el pseudocodi de l'aplicació.

8.4 Disseny de l'aplicació

En aquesta secció plantejarem el pseudocodi de l'aplicació per assolir l'objectiu del treball. A més, també s'explicaran quins atributs seran necessaris a la classe *ofApp*.

8.4.1 Python *script*

Una de les parts més importants d'aquest treball és la de descarregar els *shaders* de 'Shadertoy'. Per a fer-ho, es dissenyarà una aplicació en Python per poder fer una connexió HTTP i "parsejar" la web, per poder muntar el *vertex* i *fragment shader*, així com escriure un fitxer amb el tipus de textures que s'utilitzen (sampler2D o cubeMaps), el nom de la textura i el canal, si és que se n'utilitzen.

Per poder "parsejar" la web ho farem mitjançant JSON[6], d'aquesta manera s'obtindran els paràmetres que conformen la pàgina d'un *shader* a 'Shadertoy', des del nom fins el mateix codi.

```
FUNCIO get_shader (id)
  Request per connectar-se amb 'Shadertoy'
  SI obtenim resposta LLAVORS descodifiquem el JSON obtingut ALTRAMENT error
  SI JSON correcte LLAVORS continuar ALTRAMENT error

  PER totes les variables de la llista JSON FER
  // De fet, nomes hi ha una variable (una tupla)

    Obtenir informacio del shader (nom, descripcio, etc.); // Variable "info" de la tupla

    PER totes les variables de "renderpass" FER
    // De fet, nomes hi ha una variable (una tupla)
      Crear fitxer "inputs" per escriptura
      PER tots els inputs de la variable "inputs" de la tupla "renderpass" FER
      // Es construeix l'string que contindra els uniforms de les textures del fragment ↩
          shader
        var canals
        SI tipus input = "texture"
          canals += afegir sampler2D;
        ALTRAMENT // tipus input = cubeMap
          canals += afegir samplerCube
        FSI
        Escriure al fitxer "inputs" una entrada amb: tipus textura + nom textura + canal ↩
            de textura
      FPER
    FPER
  FPER
  Tancar fitxer "inputs"
  codi_shader = variable "code" de la tupla "renderpass"
  retorna codi_shader, info_shader
FFUNCIO
```

Quan la pàgina respon amb el contingut JSON, s'observa que aquest és una llista amb una tupla. Aquesta tupla conté 3 variables: *renderpass*, *info* i *ver*. La variable *renderpass* conté tota la informació tècnica relacionada amb el *shader*: codi font, *inputs*, *outputs*, etc. La variable *info* conté la informació del *shader*: descripció, nom, *likes*, *username*, visites, etc. La variable *ver* conté la versió del *shader*. Així, el que s'ha fet és "navegar" per aquestes 3 variables de la tupla per aconseguir els valors necessaris per construir el *shader*.

En particular, la variable *renderpass*, que conté una tupla on hi ha els *inputs* del *shader*, aquests *inputs* també contenen una tupla que descriu l'*input*: tipus, canal, nom, etc.

[6]JSON (*JavaScript Object Notation*) és un format lleuger per emmagatzemar i intercanviar dades. JSON s'utilitza per representar dades estructurals i intercanviar dades entre aplicacions client-servidor.

A continuació es plantejarà la funció *main* d'aquest *script*, on es cridarà la funció *get_shader* per obtenir el codi font del *shader*, els *uniforms* de les textures i el fitxer "inputs", i s'acabaran de construir el *vertex* i *fragment shader*.

```
FUNCIO main
  Comprovar arguments entrats // S'entra el codi del shader per comanda
  codi, info = get_shader(id)
  Crear fitxers buits pel vertex i fragment shader
  TRY
    Assignar "id" del shader com a nom dels fitxers
    Obrir els fitxers
  EXCEPT error
  Escriure "info" en el fitxer del fragment
  Escriure els uniforms per defecte en el fitxer del fragment
  Escriure els uniforms de les textures en el fitxer del fragment
  Escriure el codi font del shader en el fitxer del fragment
  Escriure la funcio main en el fitxer del fragment
  Tancar fitxer fragment
  Escriure codi del vertex shader en el fitxer vertex // Sempre es el mateix codi
  Tancar fitxer vertex
FFUNCIO
```

Finalment s'han construït el *vertex* i *fragment shader* en dos fitxer independents, més el fitxer d'"inputs" que conté els *uniforms* de les textures.

8.4.2 openFrameworks *ofApp*

Funció *setup*

La funció *setup* serà l'encarregada d'inicialitzar el sistema, a nivell de variables globals i d'interfície gràfica. Per tant, es seguirà el disseny de la GUI de la Figura 8.4.

```
FUNCIO setup()
  Inicialitzar variables globals;

  Pintar el background de l'aplicacio;

  gui = Crear UI;
  gui -> Establir un tema de colors;
  gui -> Establir el tipus de font;
  gui -> Afegir label de benvinguda;
  gui -> Afegir spacer;
  gui -> Afegir label del tipus "Entra el codi del shader";
  gui -> Afegir text input;
  gui -> Afegir label del tipus "Seleciona el tipus de render";
  gui -> Afegir llista desplegable amb 2 tipus de render (pantalla i model 3D);
  gui -> Afegir spacer;
  gui -> Afegir label del tipus "Entra el nom del model 3D";
  gui -> Afegir text input;
  // Si el render = pantalla, aquest text input s'ignora (label informatiu)
  gui -> Afegir boto de pintat; // Per executar amb els parametres entrats
  gui -> Afegir boto per canviar a pantalla completa;
  gui -> Afegir label informatiu;
  gui -> Ajustar els widgets creats;
  gui -> Establir altura de la UI;

  Afegir listener(gui,app,guiEvent);

FFUNCIO
```

Veiem que l'estructura de la funció *setup* és clara. El primer bloc serà per la inicialització de variables globals, seguidament es pinta el fons de l'aplicació, per donar més bon aspecte (per exemple, amb la funció *ofBackground*), a continuació es crea el *canvas* de la interfície gràfica

i s'hi col · loquen els diferents *widgets*, finalment s'ajusten els *widgets*, s'estableix l'alçada de la GUI i es crea el *listener* encarregat d'escoltar els *events* que passin sobre els *widgets* de la GUI.

Les variables que s'utilitzen a la funció *setup*, de les que hem vist a l'especificació de classe *ofApp*, són:

- **ofxUICanvas *gui**: *Canvas* de la interfície gràfica. Serà l'entorn on s'hi col · locaran tots els *widgets*.
- **ofxUITextInput *code**: *Widget* que fa referencia a l'entrada de text pel codi del *shader*, de 'Shadertoy', a descarregar.
- **ofxUITextInput *modeln**: *Widget* que fa referencia a l'entrada de text pel nom del model 3D a "mapejar".
- **ofxUIDropDownList *ddl**: *Widget* que fa referència a la llista desplegable on es podrà escollit el tipus de render (pantalla o model 3D). Aquest *widget* precisarà d'un vector d'*strings* amb els noms dels dos renders.

El motiu de guardar els tres *widgets* com a variables, és perquè més endavant se'n necessitarà obtenir el contingut. La resta de *widgets*, com ara *labels* o *spacers*, no fa falta guardar-los com a variables.

Funció *update*

La funció *update* serà l'encarregada d'executar l'*script* en Python (Secció 8.4.1), per descarregar el *shader* indicat, carregar les textures a partir del fitxer "inputs" i inicialitzar el *shader*. Així, el pseudocodi és:

```
FUNCIO update()
  Executar script Python amb ID shader
  Obtenir 3 llistes (tipus, noms i canals) a partir del fitxer "inputs"
  Activar la utilitzacio de coordenades normalitzades (0-1)
  Inicialitzar FBO
  SI tipus render = model 3D LLAVORS carregar model 3D

  // Carrega de textures/cubeMaps
  PER i = 0 FINS llargada llistes FER // Les 3 llistes obtingudes tenen la mateixa ↩
      llargada
    SI tipus(i) = textura LLAVORS
      Carregar la textura amb canal = canals(i)
    ALTRAMENT SI tipus(i) = cubeMap LLAVORS
      Carregar el cubeMap amb canal = canals(i)
    FSI
  FPER

  Carregar shader a partir dels fitxers obtinguts amb el Python
  Enllacar vertex i fragment shader en un sol programa

  SI tot be LLAVORS continuar ALTRAMENT error

FFUNCIO
```

La part important de la funció *update* romandrà en la comprovació del tipus de textures del fitxer "inputs". S'haurà de tenir en compte en quin canal s'han de carregar, per coincidir amb els canals de 'Shadertoy', i en el cas dels *cubeMaps* carregar correctament les 6 imatges.

Com que la funció *update* s'executa repetidament, haurà d'haver-hi una condició que s'executi el codi anterior només si s'activa el botó de pintat de la GUI.

Les variables més importants que s'utilitzen a la funció *update*, a part d'algunes de la funció *setup*, de les que hem vist a l'especificació de classe *ofApp*, són:

- **vector<string>* types, names i chan**: Els 3 vectors que contindran els tipus, els noms i els canals de les textures llegides del fitxer "inputs".
- **ofFbo fbo**: *Frame buffer object* on es renderitzarà el *shader* per posteriorment ser "pintat" a pantalla o "mapejat" sobre el model 3D.
- **string render**: *string* que indica el tipus de render que s'ha escollit a la llista desplegable. El valor s'actualitza a la funció *guiEvent*.
- **ofTexture tex0_3**: Són les 4 possibles textures que pot contenir el *shader* de 'Shadertoy'.
- **ofxCubeMap cubemap0_3**: Són els 4 possibles *cubeMaps* que pot contenir el *shader* de 'Shadertoy'.
- **ofShader shader**: *Shader* de 'Shadertoy' carregat com a variable *ofShader* de openFrameworks.

La suma de textures i cubeMaps com a màxim serà 4, però com pot haver-n'hi tant 4 d'un tipus com de l'altre, s'han de declarar totes. La funció *update* ja controlarà quines s'han de carregar i en quin canal.

Funció *draw*

Un cop inicialitzada l'aplicació, i carregades textures i *shader*, la funció *draw* s'encarregarà d'enviar al *shader* els *uniforms* que necessita i el renderitzarà en el FBO, per posteriorment ser "pintat" a pantalla o "mapejat" sobre el model 3D. Així, el pseudocodi queda:

```
FUNCIO draw()
  Activar FBO
  Activar shader
  Passar resolucio pantalla i temps actual pels uniforms: iResolution i iGlobaltTime
  SI mousePressed() LLAVORS enviar coordenades ratoli per l'uniform iMouse
  PER i = 0 FINS llargada llistes FER // Les 3 llistes obtingudes tenen la mateixa ↩
      llargada
    SI tipus(i) = textura LLAVORS
      Passar textura_i per l'uniform iChannel_i
    ALTRAMENT SI tipus(i) = cubeMap LLAVORS
      Passar cubeMap_i per l'uniform iChannel_i
    FSI
  FPER

  Renderitzar shader en un rectangle
  Desactivar shader
  Desactivar FBO
  Invertir pixels del FBO

  SI render = model 3D LLAVORS
    "mapejar" el FBO sobre del model carregat
  ALTRAMENT
    "pintar" a pantalla
  FSI
FFUNCIO
```

Cal notar la peculiaritat de "Invertir píxels del FBO". Això es fa per la diferència amb l'origen de coordenades entre OpenGL i openFrameworks. Com que l'origen d'OpenGL és l'extrem inferior esquerra, i el d'openFrameworks el superior esquerra, el renderitzat del *shader* queda invertit en comparació amb 'Shadertoy'. Per resoldre-ho, s'aplica una tècnica de mirall en els píxels emmagatzemats dins el FBO.

Com que la funció *draw* s'executa repetidament, haurà d'haver-hi una condició que s'executi el codi anterior només si s'activa el botó de pintat de la GUI i s'ha carregat correctament el *shader* a la funció *update*.

Algunes de les variables, que hem vist a l'especificació de classe *ofApp*, que s'utilitzen a la funció *draw*, i que encara no s'han vist, són:

- **ofTexture ren2tex**: Després d'aplicar la tècnica del mirall al FBO, la textura resultant es guardarà a *ren2tex*.
- **bool event1_2**: *event1* serà cert si hi ha hagut un *event* en el botó de pintat de la GUI, en canvi *event2* serà cert si s'ha executat el codi de la funció *update*.
- **ofxAssimpModelLoader model**: Variable que guardarà el model 3D carregat.
- **ofPoint rot**: Si s'arrossega el ratolí amb un dels botons premuts, *rot* contindrà les coordenades del ratolí per poder rotar el model.
- **double xaxis, zaxis**: Variables que inicialitzaran la posició del model al centre de la finestra, i que permetran moure el model.

Funcions controladores d'*events*

De les funcions controladores d'*events* que es poden implementar a la classe ofApp, caldrà implementar les següents:

```
FUNCIO keyPressed(int key)
  SI key = 'a' LLAVORS
    moure model a l'esquerra
  ALTRAMENT SI key = 'd' LLAVORS
    moure model a la dreta
  ALTRAMENT SI key = 'w' LLAVORS
    allunyar model
  ALTRAMENT SI key = 's' LLAVORS
    apropar model
  FSI
FFUNCIO

FUNCIO mouseDragged(int x,int y,int button)
  Actualitzar variable 'rot' (x,y)
FFUNCIO

FUNCIO windowResized(int w,int h)
  gui -> Establir nova altura (h)
FFUNCIO

FUNCIO exit()
  Eliminar GUI
FFUNCIO

FUNCIO guiEvent(ofxUIEventArgs &e)
  SI ID de 'e' == boto de "pintat" LLAVORS
    event1 = true
```

```
    ALTRAMENT SI ID de 'e' == llista desplegable LLAVORS
      Actualitzar tipus de render
    ALTRAMENT SI ID de 'e' == palanca pantalla completa LLAVORS
      Canviar a pantalla completa, o treure pantalla completa
    FSI
  FFUNCIO
```

Caldrà implementar tres funcions, més les dues de l'*addon ofxUI*:

- **keyPressed(key)**: Per poder moure el model amb les tecles WASD.
- **mouseDragged(x,y,button)**: Per poder rotar el model amb el ratolí.
- **windowResized(w,h)**: Per reconfigurar l'altura de la GUI si canvia la mida de la finestra.
- **exit()**: Per eliminar la GUI quan es tanqui l'aplicació.
- **guiEvent(e)**: Per controlar els *events* sobre els *widgets* de la GUI.

8.5 Diagrama de seqüència Usuari-Aplicació

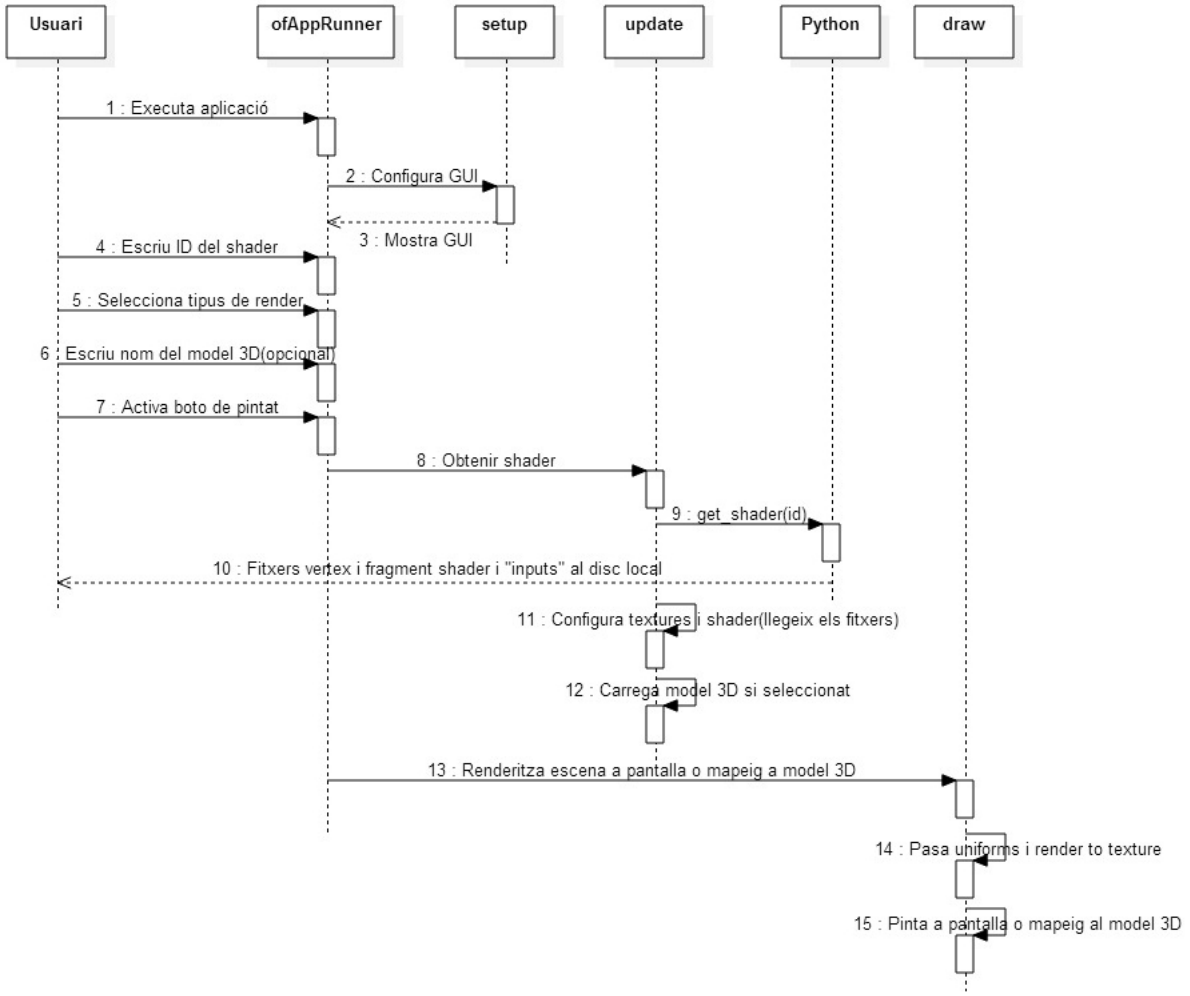


Figura 8.6: Diagrama de seqüència entre l'usuari i l'aplicació

Els missatges i accions que es duen a terme durant la interacció usuari i aplicació són els següents:

1. L'usuari executa l'aplicació i, per tant, s'executa la classe *ofAppRunner* d'openFrameworks.
2. La classe *ofAppRunner* crida la funció *setup* de la classe *ofApp* de l'aplicació, i aquesta configura la interfície gràfica.
3. L'usuari escriu i selecciona els paràmetres necessaris de la interfície gràfica.
4. La classe *ofAppRunner* crida la funció *update* de la classe *ofApp* de l'aplicació, i aquesta executa el Python per obtenir el *shader* indicat per l'usuari.
5. L'*script* en Python descarrega en el disc local de l'usuari el *shader* especificat.
6. Es continua l'execució de la funció *update*, on es configura el *shader* i es carrega el model 3D (si ho ha demanat l'usuari).
7. La classe *ofAppRunner* crida la funció *draw* de la classe *ofApp* de l'aplicació, per renderitzar l'escena.
8. La funció *draw* passa el *uniforms* al *shader*, executa el *render to texture*, i "pinta" a pantalla o "mapeja" el model 3D (segons l'elecció de l'usuari).

Tant la funció *update* com *draw* són cridades repetidament per la classe *ofAppRunner*.

Capítol 9

Implementació i proves

En aquest capítol s'explicaran els passos que s'han seguit per desenvolupar l'aplicació, des de l'*script* Python fins a les classes d'openFrameworks, així com les peculiaritats de la interfície gràfica. Tenint en compte els pseudocodis del capítol anterior, desenvoluparem els codis en el llenguatge corresponent, veurem exemples pas a pas i s'explicaran les problemàtiques que poden anar sortint. S'intercalarà la implementació dels diferents apartats amb les proves, per entendre'n bé el funcionament.

Així doncs, comencem veient la implementació de l'*script* Python per descarregar *shaders* de 'Shadertoy'.

9.1 Python *script*

Com hem vist a la Secció 8.4.1, per descarregar *shaders* de 'Shadertoy' es farà mitjançant JSON a partir de connexions HTTP. Per poder fer-ho s'utilitzaran tres llibreries:

- **json**: La llibreria *json*, entre els mòduls que incorpora, permet codificar i descodificar missatges JSON, és a dir, si volem enviar un missatge a una URL s'haurà de codificar un *string*, en canvi si rebem un missatge d'una URL s'haurà de descodificar. Aquesta llibreria incorpora aquestes funcions que ens permetran comunicar amb la pàgina web de 'Shadertoy'.

- **urllib.request**: Aquesta llibreria defineix les funcions principals que permeten obrir URLs (normalment HTTP).

- **urllib.parse**: Aquesta llibreria defineix una interfície estàndard per trencar URLs en components (adreça, localització de la xarxa, *path*, etc.), per combinar-los en un *string* URL, convertir una URL relativa en una d'absoluta, etc.

Per últim, també s'utilitzarà la llibreria *sys* que permet utilitzar comandes del sistema dins l'*script* Python.

Així doncs, comencem veient el codi de la funció *get_shader(id)* de l'*script* Python, desgranat en parts per entendre-ho millor:

```
def get_shader (id):
    url = 'https://www.shadertoy.com/shadertoy'
    headers = { 'Referer' : 'https://www.shadertoy.com/' }
    values  = { 's' : json.dumps ({'shaders' : [id]}) }

    data = urllib.parse.urlencode (values).encode ('utf-8')
    req  = urllib.request.Request (url, data, headers)
    response = urllib.request.urlopen (req)
    shader_json = response.read ().decode ('utf-8')
```

En aquest primer bloc es construeix la URL de 'Shadertoy', la qual se li enviarà la petició HTTP. Seguint la API REST[1] de 'Shadertoy', se li passen les capçaleres i les variables.

Un cop construït, es codifica i es fa la petició. 'Shadertoy' contestarà la petició amb un contingut JSON, seguint la API. Tot i això, alguns *shaders* són privats i la resposta serà buida. Ho haurem de controlar en el codi. El contingut JSON rebut de la resposta té la següent estructura:

```
[{
  "ver":"0.1",
  "info":{
    "id":"4ts3DH",
    "date":"1421822198",
    "viewed":1558,
    "name":"[2TC 15] Flying",
    "username":"iq",
    "description":"...",
    "likes":10,
    "published":3,
    "flags":0,
    "tags":["3d","raymarching","tweet"],
    "hasliked":0
  },
  "renderpass":[{
    "inputs":[{
      "id":2,
      "src":"\/presets\/tex01.jpg",
      "ctype":"texture",
      "channel":0
    }],
    "outputs":[{
      "channel":"0",
      "dst":"-1"
    }],
    "code":"...",
    "name":"",
    "description":"",
    "type":"image"
  }]
}]
```

En aquest moment *shader_json* conté la resposta JSON anteior. A continuació la descodificarem i en comprovarem el contingut:

```
    j = json.loads (shader_json)
    assert (len (j) == 1)
```

Per descodificar una resposta JSON es pot fer amb la comanda *loads*, la qual converteix l'estructura JSON en un objecte en Python, en aquest cas una llista amb una tupla. Per comprovar que s'ha rebut bé aquest contingut (per exemple el cas del *shaders* privats), es pot fer mirant la llargada de la llista, si aquesta val 1 significa que conté la tupla.

[1] REST (REpresentational State Transfer) és el principi d'arquitectura subjacent d'una web.

Com s'ha vist, el contingut de la tupla retornada són tres variables: *var*, *info* i *renderpass* (segons l'API de 'Shadertoy'). La variable a tenir en compte és la *renderpass*, que és una llista amb una tupla, que conté tot el relacionat amb el *shader*: nom, codi, *inputs*, *outputs*, etc. Per tant, s'haurà de recórrer aquesta tupla per obtenir les dades.

Es sap que s'ha de construir el *vertex* i *fragment shader*. L'estructura del fragment és:

```
Declaracio dels 'uniforms'

Codi del 'fragment shader', mes funcio 'mainImage' que calcula el color

Funcio 'main' on es crida la funcio 'mainImage'
```

On uns quants *uniforms* són sempre els mateixos, però els de les textures s'hauran d'esbrinar a partir dels *inputs* del *renderpass*. El codi es podrà obtenir directe del *renderpass*, i el *main*, per defecte, també serà sempre el mateix.

Així doncs, la idea és escriure dos fitxers, el contingut dels quals serà especificat a partir d'*strings*. L'*string* que s'haurà de construir és el dels *uniforms* de les textures, tota la resta seran variables globals ja inicialitzades. Veiem com s'ha fet:

```
  for s in j: # Recorrem la llista "j" (nomes 1 element, una tupla)
        for p in s["renderpass"]: # Recorrem la llista "renderpass" (nomes 1 element, una ↩
            tupla)
            inputs = open ("inputs", "w") # Creem l'arxiu input
            global channels # Inicialitzem una variable global que contindra l'string dels ↩
                uniforms de les textures
            for i in p["inputs"]: # Recorrem els inputs, de la variable "inputs", del "↩
                renderpass"
                if i["ctype"] == "texture": # Si el tipus de l'input es textura, l'afegim ↩
                    com a tal a "channels"
                    channels += "uniform sampler2D iChannel" + str(i["channel"]) + ";\t\t\t ↩
                        // input channel. XX = 2D/Cube\n"
                else: # Si el tipus de l'input es cubeMap, l'afegim com a tal a "channels"
                    channels += "uniform samplerCube iChannel" + str(i["channel"]) + ";\t\t\↩
                        t // input channel. XX = 2D/Cube\n"
                tex = i["src"].split('/').pop() # Afegim la textura/cubeMap a l'arxiu "↩
                    inputs"
                inputs.write(i["ctype"] + ',' + tex + ',' + str(i["channel"]) + "\n")
            inputs.close()
            channels += "\n"
            code = (p["code"]) # Obtenim el codi del shader
    return code # Retornem el codi
```

La variable "channels" és un *string* que conté els *uniforms* de les textures i els *cubeMaps* amb el canal on es troben.

El contingut del fitxer "inputs" és, per cada fila, la informació de la textura o el *cubeMap*, que, posteriorment, serà llegit des de l'aplicació d'openFrameworks. Per exemple, d'un *shader* s'ha generat l'arxiu "inputs" amb 3 textures:

```
texture,tex05.jpg,2
cubemap,cube02_0.jpg,0
cubemap,cube03_0.png,1
```

L'estructura és *tipus, nom_textura, canal*. Les variables *ctype*, *src* i *channel* són les corresponents a aquest contingut, segons l'API de 'Shadertoy', i que es pot veure en el codi anterior.

La inicialització de la resta d'*strings* que s'escriuran en els fitxer del *vertex* i el *fragment shader* és la següent:

```
shaderdecls = """
uniform vec3      iResolution;           // viewport resolution (in pixels)
uniform float     iGlobalTime;           // shader playback time (in seconds)
uniform float     iChannelTime[4];       // channel playback time (in seconds)
uniform vec3      iChannelResolution[4]; // channel resolution (in pixels)
uniform vec4      iMouse;                // mouse pixel coords. xy: current (if MLB down), ↩
    zw: click
uniform vec4      iDate;                 // (year, month, day, time in seconds)
uniform float     iSampleRate;           // sound sample rate (i.e., 44100)
"""

shadermain = """\n
void main( void ){
    vec4 color;
    mainImage( color, gl_FragCoord.xy );
    color.w = 1.0;
    gl_FragColor = color;
}
\n"""

vertex = """\
attribute vec2 pos;

void main() {
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * vec4(pos.xy,0.0,1.0);
}
\n"""

channels = """"""
```

S'han inicialitzat la resta d'*uniforms*, el *main* i el *vertex shader*. Seguidament es veurà com s'escriu als fitxers corresponents.

A continuació hi ha la implementació de la funció *main* de l'*script*, la qual cridarà la funció *get_shader(id)* i construirà els fitxers:

```
if __name__ == '__main__':
    # Comprovem que els arguments de la crida son els correctes "python getshader.py ↩
        ID_shader"
    if len (sys.argv) < 2:
        print ("Usage: %s <id>" % sys.argv[0], file=sys.stderr)

    for id in sys.argv[1:]: # Agafem l'ID_shader
        code = get_shader (id) # Cridem la funcio get_shader(id)
        code = "".join (code.split ("\r"))
        f = None # Inicialitzem els fitxers
        v = None
        while f == None or v == None:
            try: # Posem de nom als fitxers l'ID_shader i hi donem permisos
                fname = "%s.frag" % id
                vname = "%s.vert" % id
                f = open (fname, "x")
                v = open (vname, "x")

            except FileExistsError: # Si els fitxers existeixen (ja s'ha descarregat aquest ↩
                shader), no es tornen a escriure
                print ("Shader with id = %s already exists." % id, file=sys.stderr)
                sys.exit (0)

        # S'escriure el contingut als fitxers
        f.write (shaderdecls) # Uniforms per defecte
        f.write (channels) # Uniforms de textures i cubeMaps
        f.write (code) # Codi del shader
        f.write (shadermain) # Main del shader
        f.close () # Tanquem el fitxer
        v.write (vertex) # Escrivim el vertex shader
```

```
        v.close () # Tanquem el fitxer
        print ("Shader %s downloaded successfully!" % id, file=sys.stderr)
```

Seguint l'estructura del codi:

1. Es comproven els arguments de la crida de l'*script*.
2. S'agafa l'identificador del *shader* i es crida la funció get_shader(id).
3. S'inicialitzen els arxius del *vertex* i *fragment shader*.
4. Es dóna nom als arxius. Si ja es té el *shader* descarregat, no es tornen a crear els arxius.
5. Finalment s'escriuren els *strings* (*uniforms*, codi, *vertex shader*, etc.) als fitxers.

Amb aquest *script* es podrà descarregar els *shaders* de 'Shadertoy' en el disc local, d'aquesta manera des d'openFrameworks s'utilitzarà una crida al sistema per executar-lo i poder muntar els *shaders*. A continuació s'explicarà com s'ha implementat l'aplicació i la interfície gràfica, amb proves, explicant com s'han resolt les problemàtiques.

9.2 Aplicació en openFrameworks

Com s'ha vist a la Secció 8.4.2, per implementar una aplicació en openFrameworks obligatòriament s'ha d'implementar la classe *ofApp*. Aquesta classe hereta d'una altra classe *ofBaseApp*, la qual se li han de sobreescriure els mètodes. A continuació veurem com s'han implementat aquests mètodes, seguint l'anàlisi i el disseny del Capítol 8. Comencem, però, veient la classe *main* que executarà l'aplicació.

9.2.1 Funció *main*

```
#include "ofMain.h"
#include "ofApp.h"
#include "ofAppGlutWindow.h"

//========================================================================
int main( ){

  ofAppGlutWindow window;
  ofSetupOpenGL(&window, 1280, 720, OF_WINDOW);     // <-------- setup the GL context

  // this kicks off the running of my app
  // can be OF_WINDOW or OF_FULLSCREEN
  // pass in width and height too:
  ofRunApp( new ofApp());

}
```

La funció principal d'aquesta classe és crear la finestra amb *ofSetupOpenGL*, assignant-li les mides i el tipus (pantalla completa o normal). Seguidament hi executa l'aplicació *ofApp* amb *ofRunApp*. Tant un mètode com l'altre són de la classe *ofAppRunner*, que està importada a partir de l'*include* d'*ofMain.h*, que és una capçalera amb els *includes* de totes les classes d'openFrameworks. A més, també és necessari l'*include* d'*ofApp.h* per cridar el constructor, i el d'*ofAppGlutWindow.h* per declarar la finestra.

Vista la implementació de la classe *main*, entrem en el gruix de l'aplicació.

9.2.2 openFrameworks *ofApp*

Un cop s'executa la funció *ofRunApp*, la classe *ofAppRunner* comença a cridar els mètodes de l'aplicació, i si durant l'execució hi ha *events*, els detecta. El primer que fa el motor d'*ofAppRunner* és cridar la funció *setup* de l'aplicació *ofApp*, i seguidament les funcions *update* i *draw* repetidament. A continuació veurem, per parts, com s'han implementat aquests mètodes principals, més els detectors d'*events*.

Funció *setup*

En aquesta aplicació, com s'ha explicat, s'ha utilitzat aquest mètode per la inicialització de variables i de la interfície gràfica. Veiem, per parts, com s'ha implementat:

```
void ofApp::setup(){
  cout << "Setting up ..." << endl;

  // Initialization
  event1 = false;
  event2 = false;
  render = "Default";
  xaxis = ofGetWidth()/2;
  zaxis = -ofGetHeight()/2;


  // GUI set-up
  ofBackground(50,50,50,255);
  gui = new ofxUICanvas();        //Creates a canvas at (0,0) using the default width
  gui->setTheme(OFX_UI_THEME_PEPTOBISMOL);
  gui->setFont("fonts/ag-helvetica-bold-35361.ttf");
  gui->addLabel("WELCOME TO ShaderMap",OFX_UI_FONT_LARGE);
  gui->addSpacer(275,5);
```

En aquesta primera part de codi, s'inicialitzen variables importants pel comportament de l'aplicació: els *flags*, el tipus de render (per defecte es "pinta" a pantalla) i les variables que permetran moure el model (si el render seleccionat és "mapejar" sobre un model 3D). A continuació, es configura la interfície gràfica.

1. S'estableix un color RGBA pel fons de la finestra i es crea el *canvas* per la interfície gràfica amb *ofxUICanvas()*.
2. S'escull un tema (combinació de colors), amb *setTheme*. OFX_UI_THEME_PEPTOBISMOL és un enter que correspon a un tema.
3. S'escull el tipus de font que s'utilitzarà a la GUI, que es carrega d'un fitxer *.ttf*, i s'afegeix el primer *widget*, que és una etiqueta donant la benvinguda a l'aplicació.
4. S'afegeix un widget "Spacer" de 275x5 píxels.

```
  // Shader code (text input)
  gui->addLabel("Enter shaderToy code:",OFX_UI_FONT_SMALL);
  code = gui->addTextInput("CODE","",-1);

  // Drop down list
  vector<string> ls;
  ls.push_back("Default");
  ls.push_back("Load model");
  ddl = gui->addDropDownList("Select Render",ls);
  ddl->setShowCurrentSelected(true);
  gui->addSpacer(0,35);
```

```
  // 3D Model name (text input)
  gui->addLabel("Enter model name:*", OFX_UI_FONT_SMALL);
  modeln = gui->addTextInput("MODEL","",-1);
```

En aquesta segona part, es configura la part del *render*: introducció del codi del *shader* a descarregar, creació de la llista desplegable per escollir el *render*, i introducció del nom del model 3D (opcional).

5. S'afegeix una etiqueta indicant que el següent *widget* és per entrar el codi del *shader*. S'afegeix el *widget* "TextInput", i guardem el text a la variable *code*.

6. S'afegeix una llista desplegable (vector amb dos *strings*), per poder seleccionar el render, i es guarda a la variable *ddl*. A més, s'activa la visualització del render seleccionat. Finalment s'afegeix un "Spacer" de només alçada, ja que sinó, un cop desplegada la llista, es sobreposaria amb els següents *widgets*.

7. S'afegeix una etiqueta indicant que el següent *widget* és per entrar el nom del model 3D. S'afegeix el *widget* "TextInput", i guardem el text a la variable *modeln*.

```
  // Final button which draws the shader
  gui->addLabelToggle("DRAW!",false,100,30);

  // Toggle to set fullscreen
  gui->addToggle("FULLSCREEN", false);
  gui->addLabel("*only if selected render is a model", OFX_UI_FONT_SMALL);

  gui->autoSizeToFitWidgets();
  gui->setHeight(ofGetHeight());
  ofAddListener(gui->newGUIEvent, this, &ofApp::guiEvent);

}
```

Finalment s'afegeix el botó per "pintar", la palanca que permet canviar la finestra a pantalla completa, es configura la mida final de la GUI, i es crea el *listener* que s'encarregarà d'escoltar els *events* que hi hagi a la GUI.

8. S'afegeix el botó que activarà els *flags* inicialitzats anteriorment, per executar el codi de les funcions *update* i *draw*. Les mides són 100x30 píxels.

9. S'afegeix la palanca que permetrà canviar la finestra a pantalla completa. Inicialment serà *false*, és a dir, sense pantalla completa.

10. Executem el mètode *autoSizeToFitWidgets* que configura la GUI en funció dels *widgets* col · locats, a més se li configura la mateixa alçada que la finestra amb *setHeight*.

11. Finalment s'afegeix el *listener*, que escoltarà GUI *events*, i que els tractarà dins la funció *guiEvent*.

El resultat que s'obté es pot veure a la Figura 9.1. Com exemple, es mostra com s'entra el codi del shader, es selecciona el render dins la llista desplegable i s'entra el nom del model 3D.

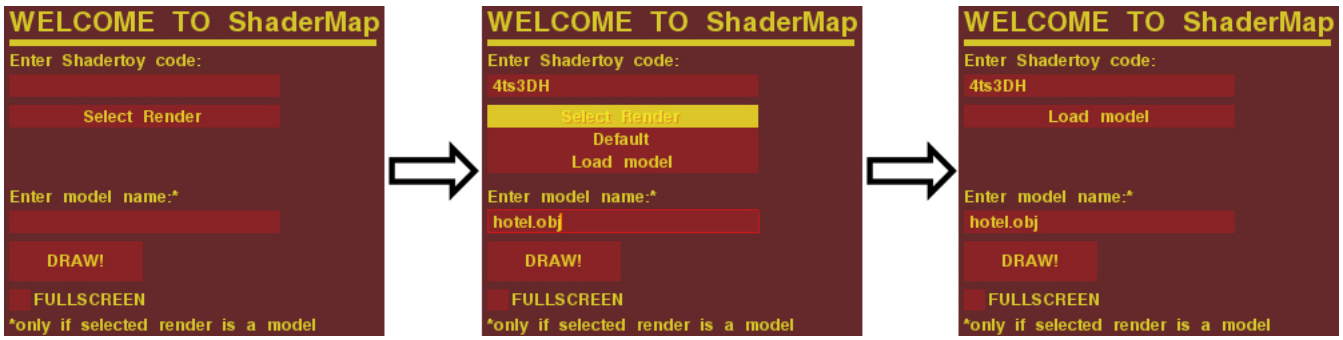


Figura 9.1: Visualització final de la interfície gràfica de l'aplicació.

En aquest moment els *events* que pugui capturar el *listener* no es tracten. A continuació veurem com s'implementa el mètode guiEvent, que és qui rep el que escolta el *listener*.

```
void ofApp::guiEvent(ofxUIEventArgs &e){
  // Activate flag 'event1' if DRAW button is pressed
  if(e.getName() == "DRAW!"){
      event1 = true;
  }
  // Desactivate flags and get the selected render
  else if(e.getName() == "Select Render"){
      event1 = false;
      event2 = false;
      ofxUIDropDownList *ddlist = (ofxUIDropDownList *) e.widget;
      vector<ofxUIWidget *> &selected = ddlist->getSelected();
      if(selected.size() > 0) render = selected[0]->getName();
  }
  // Change screen mode acording to the toggle widget
  else if(e.getName() == "FULLSCREEN"){
    ofxUIToggle *toggle = (ofxUIToggle *) e.widget;
    ofSetFullscreen(toggle->getValue());
  }
}
```

Expliquem, pas per pas, què significa cada sentència *if*:

1. Si es detecta que hi ha hagut un *event* en el *widget* del botó de "pintar", s'activa el *flag event1*. Aquest *flag* permetrà executar el codi de la funció *update*, ja que aquest funció es crida repetidament, i només interessa que s'executi el codi quan l'usuari prem el botó.

2. Si es detecta que hi ha hagut un *event* en el *widget* de la llista desplegable, es reinicialitzen els *flags* a *false*, ja que es pot estar "pintant" un nou *shader*, o el mateix en un nou model. El *flag event2* es posa a *true* quan s'ha acabat d'executar el codi d'*update* i s'ha d'executar el codi de *draw*, pel mateix motiu que es crida repetidament, però només interessa que s'executi el codi si s'ha executat el codi d'*update*. A més, s'actualitza la variable *render* amb el nou tipus de *render* a utilitzar a l'hora de "pintar".

3. Si es detecta que hi ha hagut un *event* en el *widget* de la palanca *fullscreen*, s'activa o desactiva la pantalla completa amb *ofSetFullscreen*, depenent si s'ha activat o desactivat la palanca.

Per últim només queda implementar la funció *exit*, que eliminarà la GUI quan es tanqui l'aplicació.

```
void ofApp::exit(){
    // Delete user interface when app is closed
    delete gui;
}
```

N'hi ha prou utilitzant l'operador *delete* de C++ sobre la GUI.

Vista la implementació de la funció *setup*, a continuació s'explicarà com s'ha implementat la funció *update*, que haurà de tenir en compte el *flag event1* per executar-se.

Funció *update*

La funció *update* és cridada repetidament pel motor d'openFrameworks (ofAppRunner). Tot i això, en aquesta aplicació, només interessa que s'executi quan es "pinti" un nou *shader*, ja que durant l'execució d'aquest no hi haurà canvis que s'hagin de tractar dins la funció *update*. Per això, s'utilitzarà un *flag event1* que es posarà a cert quan s'acabi de fer el *setup*, i es podrà executar el codi d'*update*, però en acabar d'executar-se es tornarà a posar a fals. Veiem el codi pas a pas:

```
void ofApp::update(){
  if(event1){

    // Execute python script
    string command;
    command = "python getshader.py " + code->getTextString();
    system(command.c_str());

    // Read inputs file
    types = new vector<string>();
    names = new vector<string>();
    chan = new vector<string>();
    ofFile file;
    file.open(ofToDataPath("inputs"), ofFile::ReadWrite, false);
    ofBuffer buff = file.readToBuffer();
    while(!buff.isLastLine()){
        vector<string> line;
        line = ofSplitString(buff.getNextLine(),",");
        types->push_back(line.at(0));
        names->push_back(line.at(1));
        chan->push_back(line.at(2));
    }
```

En aquest primer bloc de codi s'executa l'*script* en Python per descarregar el *shader*, i s'obté el contingut del fitxer "inputs" generat per l'*script*.

1. Es construeix la comanda del sistema per executar un *script* Python, en aquest cas el Python a executar és l'*script* per descarregar *shaders* "getshader.py". El shader a descarregar el dóna el *widget* "code" de la GUI, que se n'obté el contingut amb *getTextString*.

2. Un cop s'ha descarregat el *shader*, llegim el fitxer "inputs" que s'ha generat i guardem les textures (tipus, nom i canal) en tres vectors diferents. Per exemple: llegim la primera línia del fitxer i tenim "texture,tex01.jpg,0", llavors la posició 0 de *types* contindrà un tipus "texture", *names* contindrà "tex01.jpg" i *chan* contindrà '0'.

Per poder obrir el fitxer s'utilitza la classe *ofFile*, que permet especificar la ruta del fitxer i el mode de lectura/escriptura. Per llegir-lo es pot fer amb la classe *ofBuffer*.

```
// Load textures with normalized texcoords (0..1)
ofDisableArbTex();
ofEnableNormalizedTexCoords();

// Render to Texture (frame buffer object)
fbo.allocate(ofGetWidth(), ofGetHeight(), GL_RGBA);
fbo.begin();
ofClear(255,255,255,255);
fbo.end();
if(render == "Load model"){
    model.loadModel("models/" + modeln->getTextString());
    model.enableNormals();
    model.setScale(-1,1,1);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_NORMALIZE);
}
rot.y = ofGetHeight()/2;
```

En aquest bloc s'activa la utilització de textures amb coordenades normalitzades (0-1), s'inicialitza el FBO i es carrega el model, si s'ha indicat a la GUI.

3. S'activa la utilització de coordenades normalitzades per a les textures. Per fer-ho s'utilitzen les funcions *ofDisableArbTex* i *ofEnableNormalizedTexCoords*.

4. S'inicialitza el FBO amb les mides de la finestra, i se'n neteja el contingut.

5. Si a la llista desplegable s'ha seleccionat que es "mapejarà" sobre un model 3D, es carrega aquest model a partir del contingut del *widget modeln*. A més, s'activa la utilització d'il · luminació a l'escena, per tal de crear realisme en el model, i veure'n els detalls. Si no es fes això pasaria el següent:

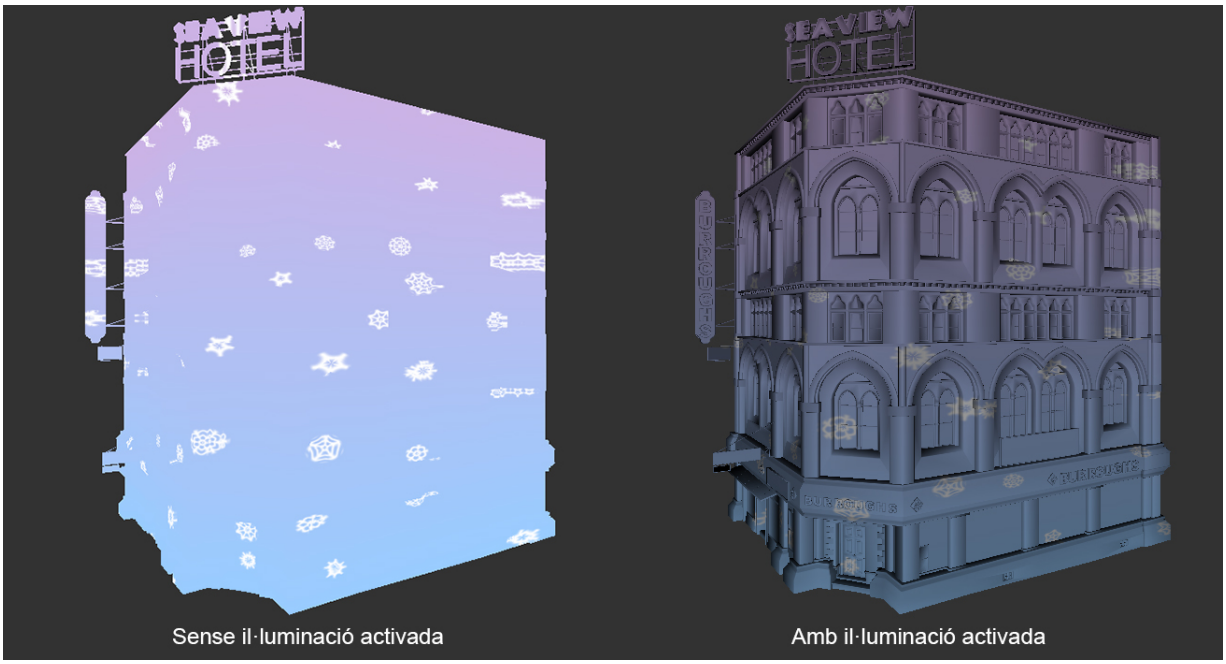


Figura 9.2: Comparativa del mateix model amb i sense il · luminació

6. S'inicialitza l'altura del model *rot.y*, al centre de la finestra. Aquesta variable ajustarà l'origen del model al centre de la finestra.

```
// Load textures acording to its channel -> texi where 'i' is the channel
  for(int i = 0; i < types->size(); i++){
      if(types->at(i) == "texture"){ // If type is 'texure'
          if(chan->at(i) == "0") // If channel is 0, load in tex0
              ofLoadImage(tex0,"textures/" + names->at(i));
          else if(chan->at(i) == "1")
              ofLoadImage(tex1,"textures/" + names->at(i));
          else if(chan->at(i) == "2")
              ofLoadImage(tex2,"textures/" + names->at(i));
          else
              ofLoadImage(tex3,"textures/" + names->at(i));
      }
      else if(types->at(i) == "cubemap"){ // Else if type is 'cubeMap'
          string t = "textures/";
          string sName = ofSplitString(names->at(i),"_").at(0);
          string ext = ofSplitString(names->at(i),".").at(1);
          ext = "." + ext;
          if(chan->at(i) == "0") // If channel is 0, load in cubemap0
              cubemap0.loadImages(t+sName+"_0"+ext,t+sName+"_1"+ext,t+sName+"_2"+ext,t+sName↩
                  +"_3"+ext,t+sName+"_4"+ext,t+sName+"_5"+ext);
          else if(chan->at(i) == "1")
              cubemap1.loadImages(t+sName+"_0"+ext,t+sName+"_1"+ext,t+sName+"_2"+ext,t+sName↩
                  +"_3"+ext,t+sName+"_4"+ext,t+sName+"_5"+ext);
          else if(chan->at(i) == "2")
              cubemap2.loadImages(t+sName+"_0"+ext,t+sName+"_1"+ext,t+sName+"_2"+ext,t+sName↩
                  +"_3"+ext,t+sName+"_4"+ext,t+sName+"_5"+ext);
          else
              cubemap3.loadImages(t+sName+"_0"+ext,t+sName+"_1"+ext,t+sName+"_2"+ext,t+sName↩
                  +"_3"+ext,t+sName+"_4"+ext,t+sName+"_5"+ext);
      }
  }
```

En aquest bloc es carreguen les imatges, que faran de textures, que s'han llegit del fitxer "inputs". Per fer-ho, s'utilitzarà la classe *ofTexture* per emmagatzemar-les, o l'*addon ofxCubeMap* si *type* és "cubemap". S'haurà de tenir en compte en quin canal es troben.

7. Per totes les textures/cubemaps que contenia el fitxer "inputs" es comprova si el *type* és "texture" o "cubemap".

8. Si és "texture" es comprova en quin canal s'haurà de carregar, i dependent d'aquest es carrega en un *ofTexture* o un altre. Per carregar una imatge en un *ofTexture* s'utilitza *ofLoadImage*.

9. Si, en canvi, el *type* és "cubemap", el fitxer "inputs" només conté una de les 6 imatges a carregar. El que es fa és obtenir el nom (traient el número 0 i l'extensió de la imatge) i es construeixen els noms de les 6 imatges "nom+_i+extensió". Es carreguen de la mateixa manera que les textures, tenint en compte el canal i, en aquest cas, utilitzant l'*addon ofxCubeMap* amb la funció *loadImages*. Com que les 6 imatges tenen el mateix nom, i nomes canvia el "_i", es pot fer així, respectant l'ordre, sempre que es tinguin les 6 imatges guardades.

```
        // If there is a running shader, unload it before load a new one
        if(shader.isLoaded())
            shader.unload();

        // Load shader
        shader.load("shaders/"+code->getTextString());
        shader.bindDefaults();
        shader.linkProgram();

        // Check if GUI parameters are correct
        cout << endl;
        if(shader.isLoaded()){
            if(render == "Load model"){
                if(model.hasMeshes())
                    cout << "Executing shader on model..." << endl;
                else{
                    cout << "Model '" << modeln->getTextString() << "' doesn't exists!" << ↩
                        endl;
                }
            }
            else{
                cout << "Executing shader ..." << endl;
            }
        }
        else{
            cout << "Shader ID is incorrect!" << endl;
        }

        event1 = false;
        event2 = true;
    }
}
```

Finalment, l'últim bloc de la funció *update* carrega el *shader*, comprovant abans que no se n'estigui executant un, i comprovarà que s'hagi carregat bé (s'ha entrat bé el codi del *shader*), i si s'ha seleccionat "mapejar" a un model 3D, que aquest també s'hagi entrat bé.

10. Si s'està executant un *shader*, es desactiva abans de carregar-ne un de nou.

11. Es carrega el nou *shader* amb els valors per defecte, i es crea el programa.

12. Es comprova que s'hagi carregat correctament el *shader* i el model (si s'ha escollit render = model).

13. Per últim es desactiva el *flag event1* (ja s'ha executat el codi d'*update*), i s'activa el *flag event2* (ja es pot executar el codi de la funció *draw*).

Funció draw

Igual que la funció *update*, la funció *draw* és cridada repetidament, precisament just després de la funció *update*. De la mateixa manera que la funció *update*, la funció *draw* només interessa que s'executi quan s'hagi executat el codi d'*update*, per això l'ús dels *flags*, en aquest cas *event2*. L'objectiu de la funció *draw* és "pintar" el resultat final, és a dir, el shader a pantalla o "mapejat" sobre un model 3D. Veiem el codi pas a pas, i n'entendrem el funcionament:

```
void ofApp::draw(){
    if(event2 && shader.isLoaded()){
        cubemap0.bindToTextureUnit(0);
        cubemap1.bindToTextureUnit(1);
        cubemap2.bindToTextureUnit(2);
        cubemap3.bindToTextureUnit(3);

        // Begin render to texture
        fbo.begin();
        ofBackground(50,50,50,255);
        shader.begin();
```

Un cop s'ha executat la funció *update* podem tenir, per exemple, dues textures que aniran als canals 0 i 1, i dos *cubeMaps* que aniran als canals 2 i 3, de la mateixa manera que podríem tenir 4 *cubeMaps*. Per això, com que podria haver-hi aquest cas, enllacem els *cubeMaps* en els canals corresponents, ja que en el cas dels *cubeMaps* s'han d'enllaçar abans de començar a passar *uniforms*. Per això, com es pot veure, es fan els *bindToTextureUnit* abans de començar a utilitzar el *shader*, tot i que després podrien no utilitzar-se. Per tant:

1. S'enllacen els possibles *cubeMaps* en els respectius canals.
2. S'activa el FBO per començar el *render to texture*. Tot el que es "pinti" abans de tancar el FBO, quedarà guardat en aquest.
3. Es "pinta" el color de fons i s'activa el *shader*.

En aquest moment el renderitzat del *shader* quedarà guardat en el FBO. Ja es pot començar a passar els *uniforms*.

```
        // Pass iResolution, iGlobalTime and iMouse uniforms to the shader
        float resolution[] = {ofGetWidth(),ofGetHeight(),1};
        shader.setUniform3fv("iResolution",resolution);
        shader.setUniform1f("iGlobalTime",ofGetElapsedTimef());
        if(ofGetMousePressed()){
            float mouse[] = {ofGetMouseX(),abs(ofGetMouseY()-ofGetHeight()),0,0};
            shader.setUniform4fv("iMouse",mouse);
        }
```

1. Es passa la resolució de la finestra a l'*uniform iResolution*.
2. Es passa el temps d'execució de l'aplicació a l'*uniform iGlobalTime*.
3. Finalment es passen les coordenades del ratolí a l'*uniform iMouse*, en el cas que s'estigui prement algun botó del ratolí.

Recordem que l'origen de coordenades del 'Shadertoy' és el punt inferior esquerra de la finestra, en canvi en openFrameworks és el superior esquerra. Això s'ha de tenir en compte quan es passa la coordenada Y del ratolí, per això s'inverteix el valor en funció de l'alçada de la finestra.

A continuació veurem com es passen els *uniforms* de les textures i els *cubeMaps*, tenint en compte els vectors, ja que així es sabrà quants n'hi ha de cada.

```
        // Pass iChanneli uniforms to the shader
        for(int i = 0; i < types->size(); i++){
            if(types->at(i) == "texture"){ // If type is 'texture'
                if(chan->at(i) == "0"){
                    tex0.setTextureWrap(GL_REPEAT,GL_REPEAT);
                    shader.setUniformTexture("iChannel0", tex0, 0);
                }
                else if(chan->at(i) == "1"){
                    tex1.setTextureWrap(GL_REPEAT,GL_REPEAT);
                    shader.setUniformTexture("iChannel1", tex1, 1);
                }
                else if(chan->at(i) == "2"){
                    tex2.setTextureWrap(GL_REPEAT,GL_REPEAT);
                    shader.setUniformTexture("iChannel2", tex2, 2);
                }
                else{
                    tex3.setTextureWrap(GL_REPEAT,GL_REPEAT);
                    shader.setUniformTexture("iChannel3", tex3, 3);
                }
            }
            else if(types->at(i) == "cubemap"){ // Else if type is 'cubeMap'
                if(chan->at(i) == "0")
                    shader.setUniform1i("iChannel0", 0);
                else if(chan->at(i) == "1")
                    shader.setUniform1i("iChannel1", 1);
                else if(chan->at(i) == "2")
                    shader.setUniform1i("iChannel2", 2);
                else shader.setUniform1i("iChannel3", 3);
            }
        }
```

Es recorre el vector de *types*. Si seguim l'exemple anterior de 2 textures i 2 *cubeMaps*, per les textures s'entrarà en els 2 primers *ifs* de *chan* 0 i 1, passant com a uniform les textures *tex0* i *tex1*, que son les que s'hauran carregat a la funció *update*. Per altra banda, pels 2 *cubeMaps* s'entrarà als *ifs* de *chan* 2 i 3 del segon bloc, que són els que s'hauran carregat a la funció *update*.

En el cas dels *cubeMaps*, només es passa un enter (el número del canal), ja que són on estan enllaçats, com s'ha vist en el bloc de codi anterior. Tot i haver enllaçat els canals 0-3, en aquest cas només s'utilitzen el 2 i 3, ja que el 0 i 1 són per les dues textures.

Cal notar, també, que abans de passar les textures com a *uniforms* se'ls hi canvia el tipus de *wrap*, per defecte GL_CLAMP_TO _EDGE, a GL_REPEAT. Això es fa perquè interessa que es repeteixi la textura per tota la geometria, i no pas tenir una textura estirada per tota la geometria per emplenar-la. Veure la Figura 9.3.

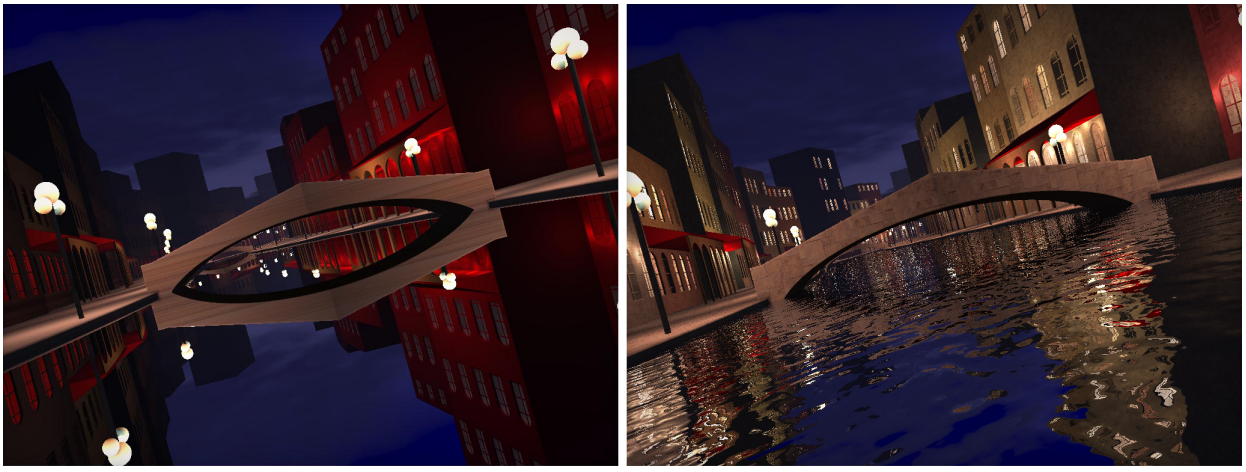


Figura 9.3: Comparativa entre utilitzar *wrap* GL_CLAMP_TO_EDGE o GL_REPEAT

Com es pot veure a la Figura 9.3, la diferència entre un tipus de *wrap* i l'altre és considerable, de fet amb el *wrap* GL_CLAMP_TO_EDGE sembla que no hi hagi textures. On es veu més la diferència és en el pont i l'aigua, a l'esquerra les textures estan estirades i a la dreta es repeteix el patró.

```
        ofRect(0, 0, ofGetWidth(), ofGetHeight());

        shader.end();

        fbo.end();

        cubemap0.unbind();
        cubemap1.unbind();
        cubemap2.unbind();
        cubemap3.unbind();
```

Un cop passats tots els *uniforms*, es renderitza el resultat en un rectangle de la mida de la finestra, es finalitza el *shader* i el FBO i es desenllacen els *cubeMaps* dels respectius canals.

```
        // Flip the texture
        ofPixels texPix;
        fbo.readToPixels(texPix);
        texPix.mirror(true,false);
        ren2tex.loadData(texPix);

        // If selected render is a model then
        if(render == "Load model"){
            model.setPosition(xaxis, rot.y, zaxis);
            model.setRotation(0,rot.x,0,1,0);
            ofEnableDepthTest();
            ren2tex.bind();
            model.drawFaces();
            ren2tex.unbind();
            ofDisableDepthTest();
        }
        else{ // Paint in window
            ren2tex.draw(0,0);
        }
    }
}
```

Per acabar, s'ha de pintar el contingut renderitzat del FBO, a la pantalla o "mapejant" el model 3D.

4. Sabem que l'origen de coordenades de l'openFrameworks és diferent al del 'Shadertoy', per tant el renderitzat del FBO haurà quedat invertit. Per revertir-ho, s'agafen els píxels guardats en el FBO i se'ls hi aplica un efecte mirall. A continuació, es guarden aquests píxels en una textura que serà la que es "pintarà" (el *shader* tal hi com es veu a 'Shadertoy').

5. Si es vol "mapejar" en un model s'activarà la textura abans de pintar les cares, d'aquesta manera les cares quedaran "mapejades" amb la textura. A més, s'activarà el *Depth Test*, que renderitza en funció de la profunditat de les cares, i no en l'ordre de "pintat".

6. En canvi, si es vol "pintar" a pantalla, directament pintarem la textura.

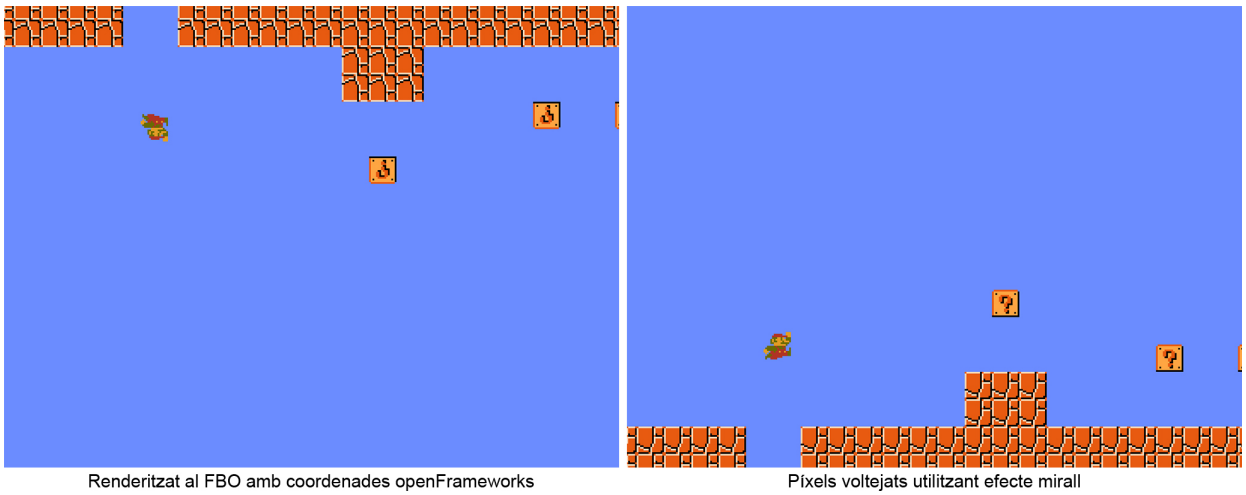


Figura 9.4: Comparativa d'abans i després d'aplicar l'efecte mirall

La imatge de l'esquerra és el resultat de "pintar" directament del FBO. Això és degut al problema del diferent origen de coordenades entre 'Shadertoy' i openFrameworks. A la dreta, el problema s'ha solucionat aplicant un efecte mirall als píxels del FBO.

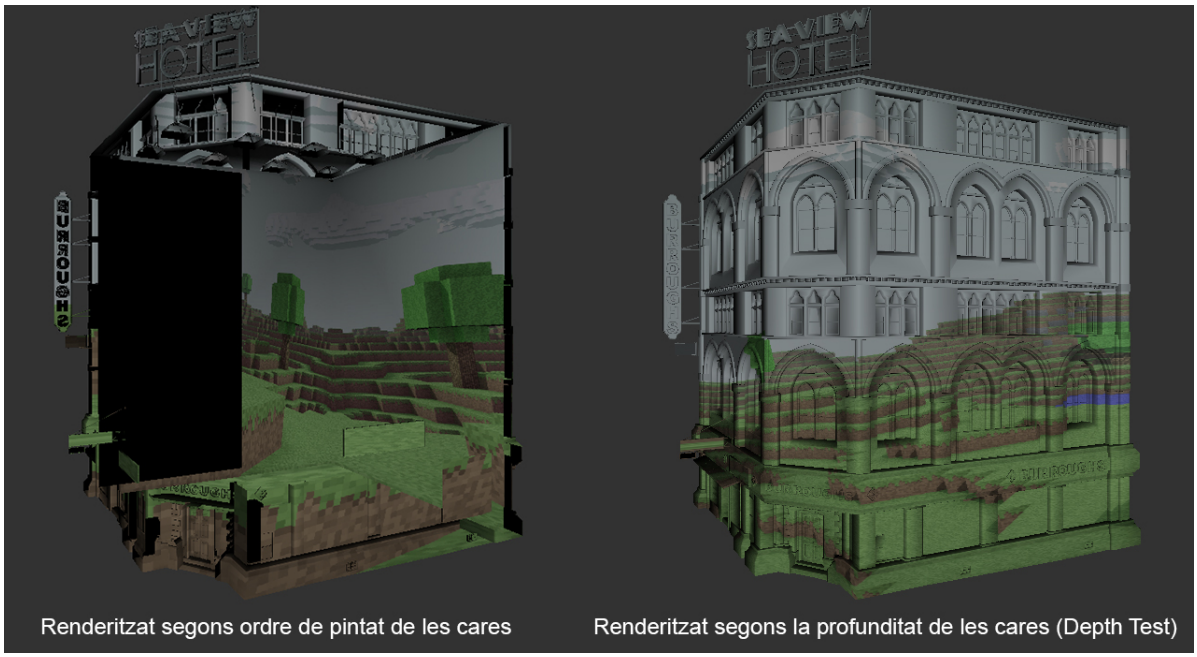


Figura 9.5: Comparativa entre renderitzar segons la profunditat de les cares o l'ordre de "pintat"

La imatge de l'esquerra és el resultat de renderitzar segon l'ordre en que es "pinten" les cares (la última pintada serà la que es veurà més a prop), en canvi la de la dreta es renderitza segons la "posició" en l'eix Z, és a dir, es veuran primer les que estiguin mes a prop.

Control dels *events*

Per últim, les tres funcions controladores d'events a implementar són les següents:

```
void ofApp::keyPressed(int key){
    double step = 10.0;
    if(key == 'a')
        xaxis = xaxis - step;
    else if(key == 'd')
        xaxis = xaxis + step;
    else if(key == 'w')
        zaxis = zaxis - step;
    else if(key == 's')
        zaxis = zaxis + step;
}

void ofApp::mouseDragged(int x, int y, int button){
    rot.x = x;
    rot.y = y;
}

void ofApp::windowResized(int w, int h){
    gui->setHeight(h);
}
```

1. A la funció *keyPressed* es modifiquen les variables que permeten moure el model utilitzant les tecles WASD.
2. A la funció *mouseDragged* es modifiquen les variables que permeten rotar el model utilitzant el ratolí.
3. A la funció *windowResized* es canvia l'altura de la GUI quan es canvia la mida de la finestra, pel cas canvi de vista a pantalla completa o viceversa.

Capítol 10

Resultats

En aquest capítol es mostrarà el grau d'assoliment dels objectius, mitjançant exemples del funcionament del projecte desenvolupat. L'objectiu d'aquest capítol serà mostrar que l'aplicació resultant del projecte fa les tasques especificades originalment.

A més, s'exposarà la validesa legal de l'aplicació, especialment la que fa referència a la Llei Orgànica de Protecció de Dades de Caràcter Personal (LOPD) i la Llei de Serveis de la Societat de la Informació i Comerç Electrònic (LSSICE).

10.1 Resultats en funció dels objectius

L'objectiu principal d'aquest treball era desenvolupar una aplicació on es poguessin previsualitzar projeccions de *projection mapping*, mitjançant la tecnologia dels *shaders* (GLSL) sobre models 3D, on els VDJs poguessin treballar amb una *pipeline* robusta. Per a fer-ho, s'ha escollit 'Shadertoy' com a plataforma de *shaders*, i openFrameworks com entorn de treball. A més, es volia que la interfície gràfica de l'aplicació fos intuïtiva i fàcil d'utilitzar.

El resultat final de l'aplicació és el de la Figura 10.1.

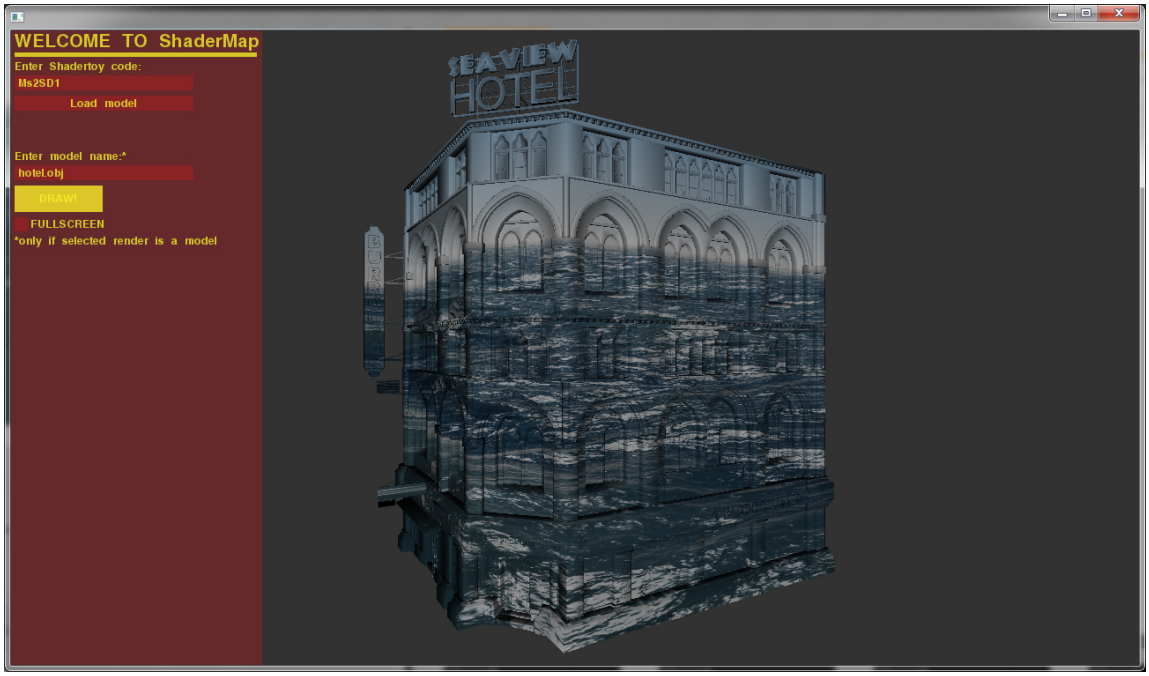


Figura 10.1: Resultat final de l'aplicació desenvolupada en aquest projecte

Analitzem-ho per parts:

1. A la part esquerra de l'aplicació es pot veure la interfície gràfica, amb uns colors que es contrasten amb la resta de l'aplicació. Havia de ser una interfície fàcil d'utilitzar, per això es demanen pocs paràmetres d'entrada per fer funcionar l'aplicació, a més que minimitza molt el mal funcionament per paràmetres erronis, tot i que l'aplicació els controla correctament i avisa si hi ha hagut problemes. Veure la Figura 10.2

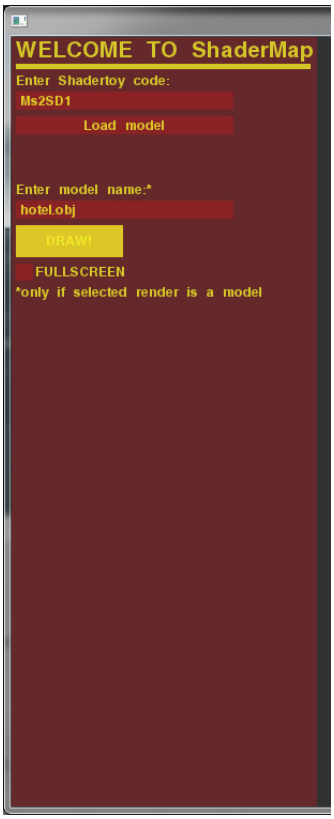


Figura 10.2: Interfície gràfica de l'aplicació resultant

Així doncs, tenim dues caixes de text (per introduir l'ID del *shader*, i per especificar el nom del model 3D), una llista desplegable per escollir el tipus de render (a pantalla o a model 3D), més el botó de "pintar" i la palanca per canviar a pantalla completa. Es demana la mínima informació pel màxim rendiment. En aquest exemple s'ha escollit el shader amb ID = *Ms2SD1*, tipus de render = "mapejar" en un model 3D, i nom del model = *hotel.obj*.

Per tant, s'ha pogut implementar una interfície gràfica que satisfà els requisits per fer funcionar l'aplicació. Aquesta interfície gràfica s'ha acabat implementant utilitzant un *addon* d'openFrameworks anomenat *ofxUI*.

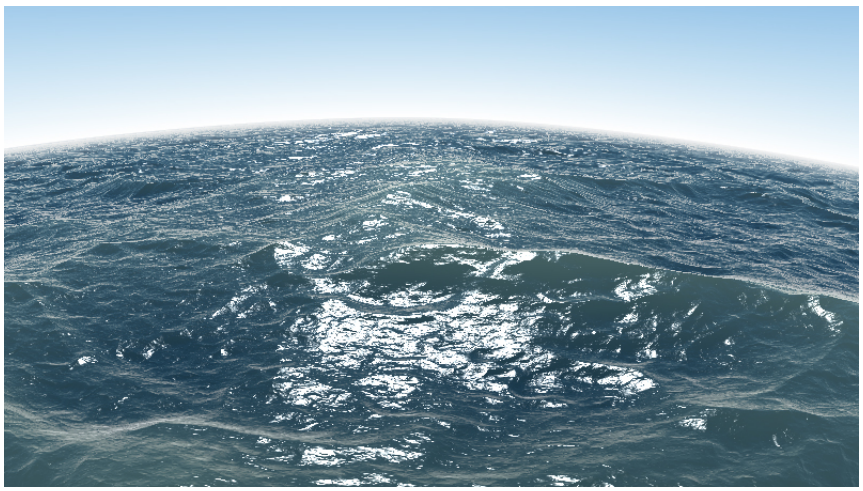


Figura 10.3: *Shader* utilitzat per dur a terme el mapeig

2. L'aplicació permet descarregar *shaders*, de tercers o de disseny propi, gràcies a un *script* en Python. Aquest *shaders* es guarden com a fitxers al disc local, per ser posteriorment executats per openFrameworks. Tot aquest procediment passa desapercebut per a l'usuari. En aquest cas el *shader* escollit és el de la Figura 10.3.

   L'obtenció dels *shaders* ha estat possible gràcies a les llibreries de connexions HTTP i d'intercanvi de missatges JSON de Python. A més de la utilització de la classe *ofShader* d'openFrameworks per executar-los.

   L'aplicació permet en qualsevol moment canviar el tipus de render i tornar a "pintar". Veure la Figura 10.4

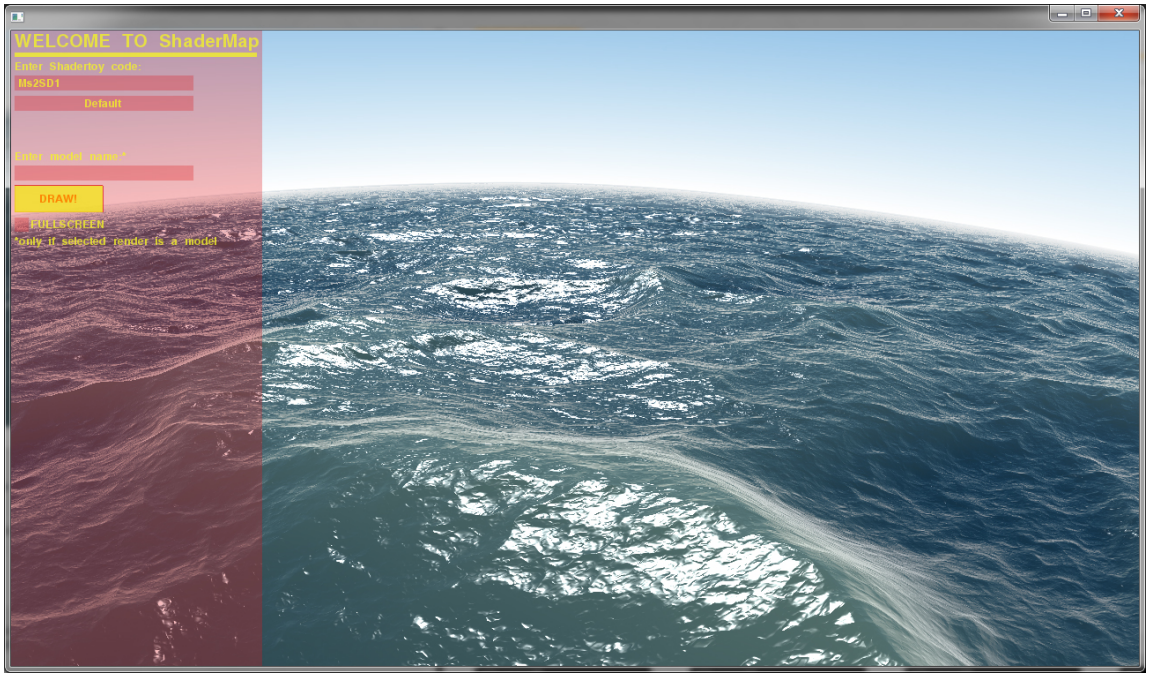


Figura 10.4: Canvi de render utilitzant "pintar" el *shader* a pantalla

La interfície gràfica no la perdem en cap moment, fins i tot quan es "pinta" a la finestra té un punt de transparència, gràcies a l'*alpha* dels colors RGBA.

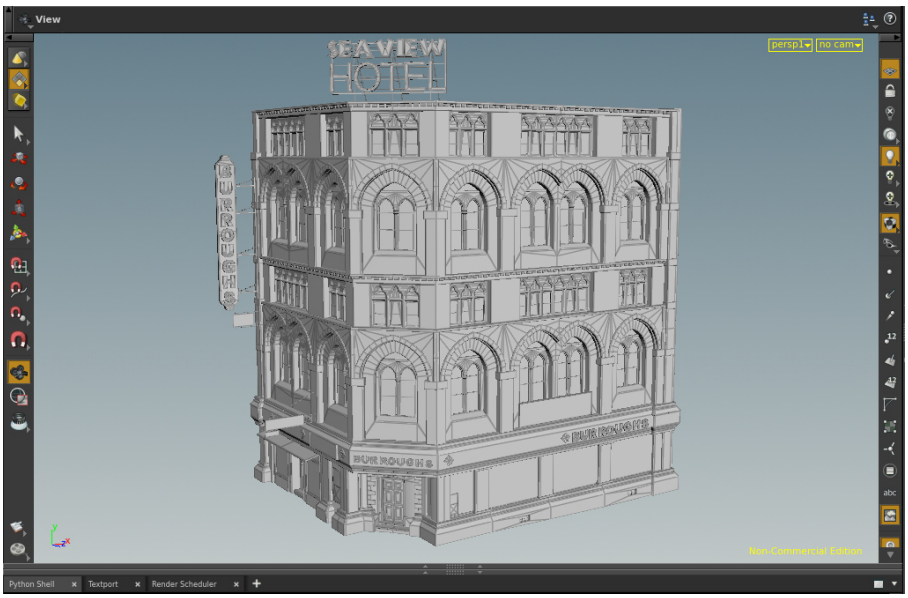


Figura 10.5: Model escollit per mapejar-hi el *shader*

3. En l'exemple de resultat final s'ha escollit un hotel com a model 3D, per a mapejar-hi el *shader*. Aquest model ha estat desenvolupat amb Houdini, però es pot carregar de qualsevol altra plataforma com Blender, o fins i tot descarregats d'Internet. L'*addon* d'openFrameworks, *ofxAssimpModelLoader*, el qual permet carregar els models, accepta molts formats de model 3D.

   L'usuari haurà d'haver definit les coordenades on voldrà veure mapejat el *shader*, de la mateixa manera com ho fan els professionals del *projection mapping*.

4. Finalment, en els cas d'escollir "mapejar" un model 3D, es té lliure accés al model, podent-l'hi canviar la posició i també rotar-lo. Cal dir que els *shaders* estan en moviment, i evidentment sobre paper no s'aprecia el cent per cent del resultat. Veure Figures 10.6 i 10.7.

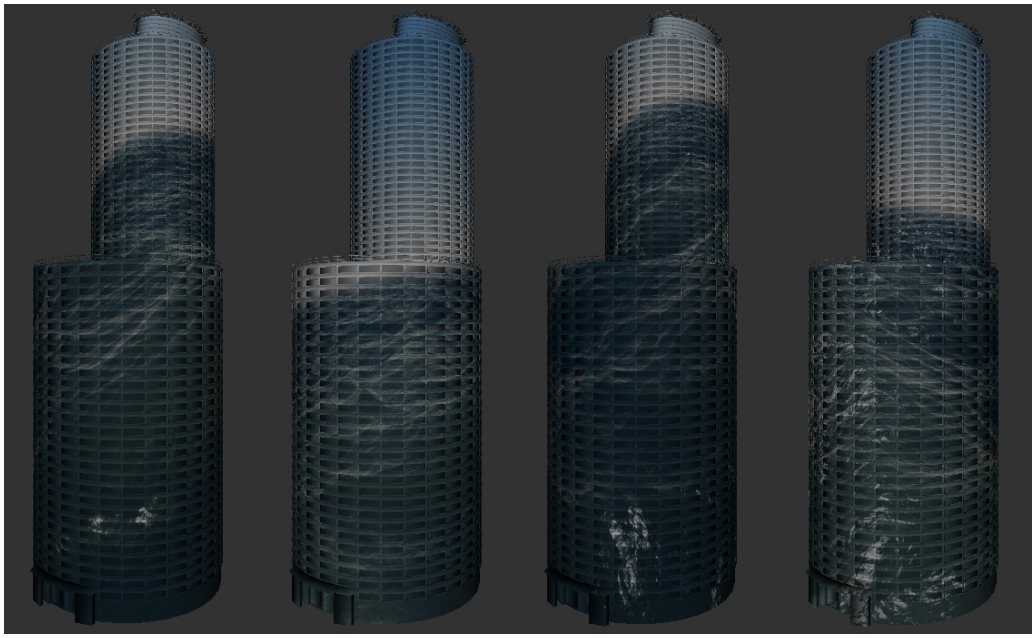


Figura 10.6: Mapeig d'un *shader* en un edifici en diferents instants de temps

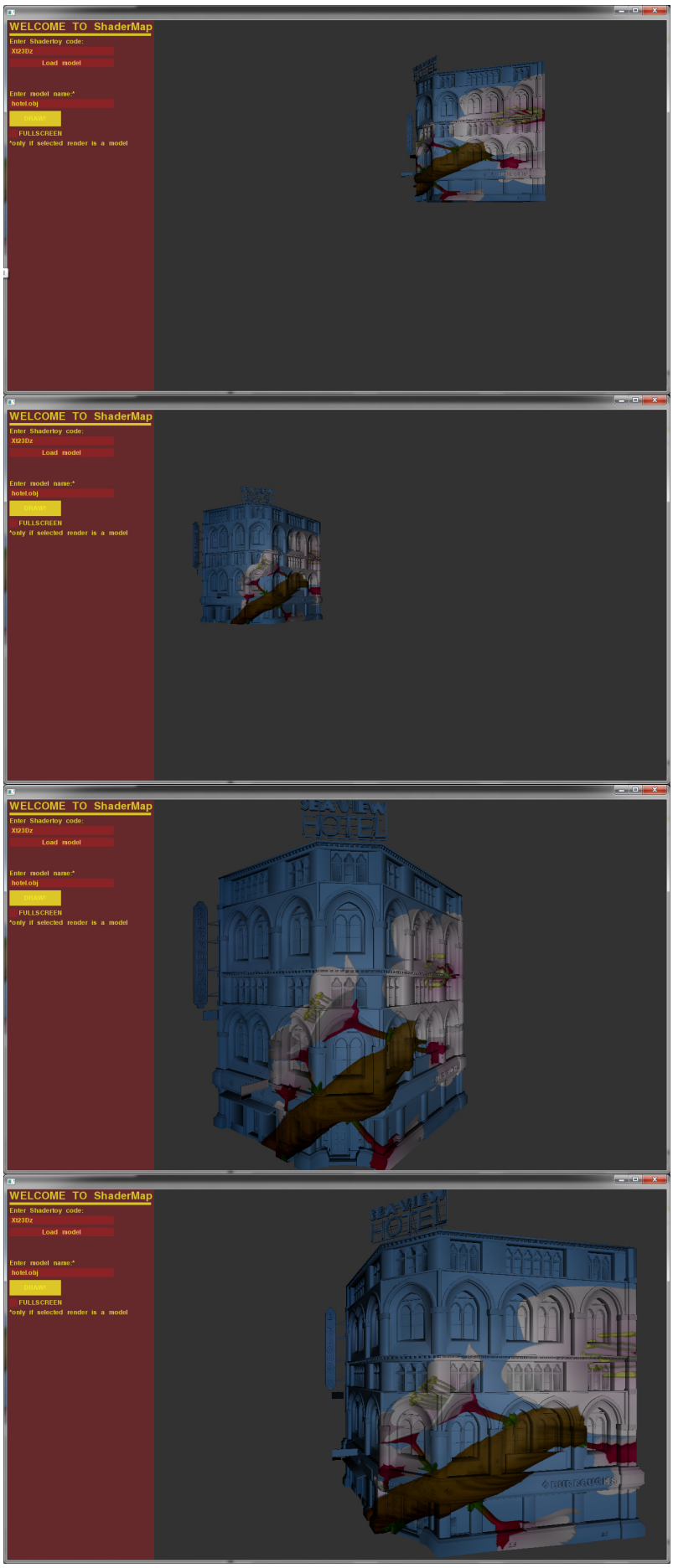


Figura 10.7: Diferents posicions del model 3D, gràcies a la possibilitat de poder moure'l i rotar-lo.

A continuació es mostren dos exemples "pintats" a la finestra, on el primer (Figura 10.8) utilitza els 4 canals de textura (amb textures), i el segon utilitza dos *cubeMaps* i una textura (Figura 10.9).

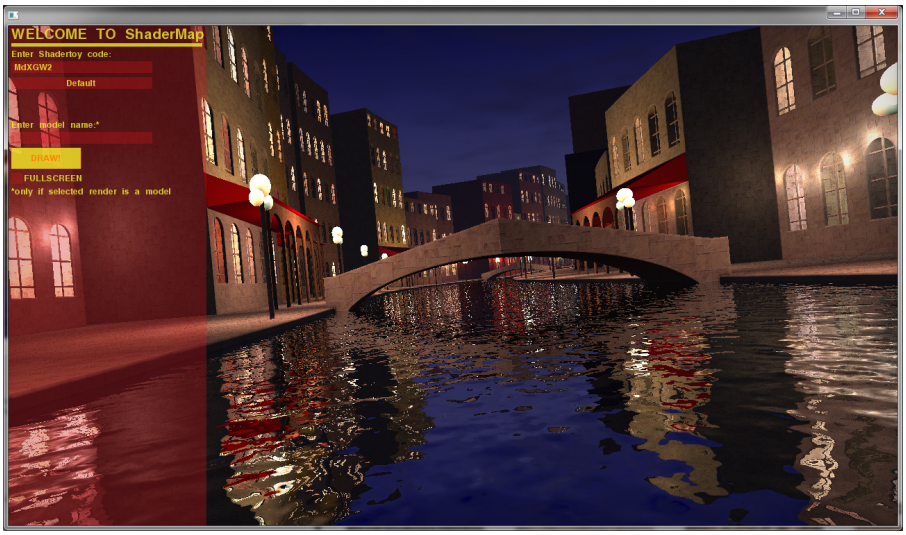


Figura 10.8: *Shader* de Venècia "pintat" a la finestra

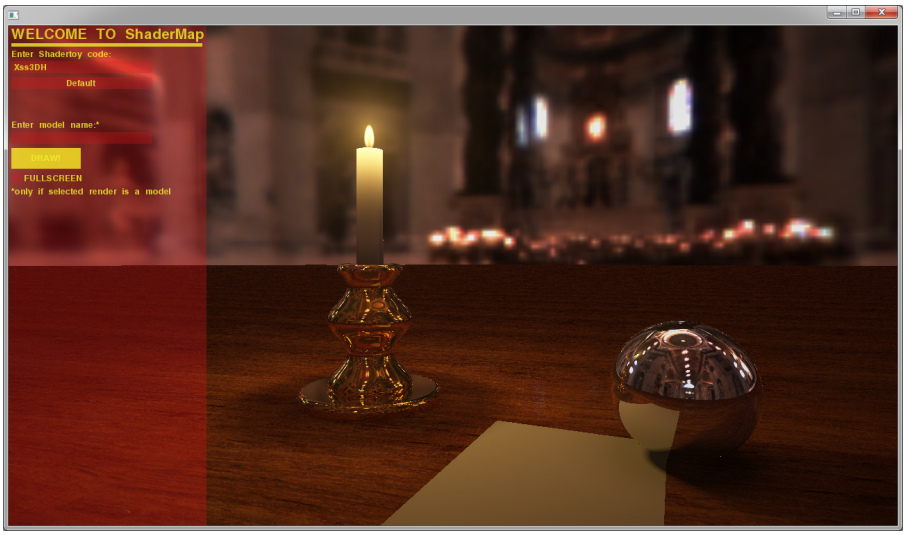


Figura 10.9: *Shader* que utilitza dos *cubeMaps* i una textura "pintat" a la finestra

10.2 Validesa legal de l'aplicació

Un aspecte a tenir en compte quan desenvolupem una aplicació és la validesa legal d'aquesta, és a dir, si compleix la legislació actual pel que fa a la Llei Orgànica de Protecció de Dades de Caràcter Personal (LOPD) i la Llei de Serveis de la Societat de la Informació i Comerç Electrònic (LSSICE).

Si parlem de la de protecció de dades, l'aplicació no emmagatzema cap tipus d'informació de l'usuari, motiu pel qual complim l'esmentat per la LOPD. Per altra banda, si ens centrem en el comerç electrònic, l'aplicació d'aquest projecte no inclou cap servei de pagament, motiu pel qual complim l'esmentat per la LSSICE.

Capítol 11

Conclusions

Per assolir els objectius d'aquest treball final de grau, es van marcar unes tasques a realitzar (veure Secció 1.3). Aquestes tasques s'han pogut dur a terme, no per això sense trobar entrebancs.

S'ha pogut dissenyar una aplicació amb una *pipeline* robusta, per a dur a terme previsualitzacions de projeccions de *projection mapping*, mitjançant la tecnologia dels *shaders* (GLSL) sobre models 3D. Per això, s'han ampliat els coneixements d'informàtica gràfica i de com modificar la *pipeline* de la GPU, a més dels conceptes teòrics d'OpenGL. També s'ha complert que totes les eines utilitzades durant el procés de desenvolupament són "Open Source", ja que es volia sortir del model d'aplicació tancada com la majoria que s'utilitzen per *projection mapping*.

S'han obert portes a tècniques i eines totalment desconegudes, com és el *projection mapping* i l'openFrameworks. La recerca del funcionament de tot aquest món no ha estat fàcil. A més, openFrameworks té un ventall molt gran de possibilitats, i en aquest projecte només se n'ha vist una part. S'ha estudiat a fons el funcionament d'una aplicació web de *shaders*, 'Shadertoy', veient com és el funcionament del seu motor i com executa els *shaders*, així com també la seva API. També s'ha pogut veure una part del funcionament d'un nou llenguatge, el Python.

L'elecció de les eines no ha estat fàcil, partint del desconeixement total sobre eines de *projection mapping*, tot i això es va poder tenir una trobada amb un expert que ens va endinsar en aquest món.

A la pràctica, no ha estat fàcil dur a terme l'aplicació, ja que s'ha hagut d'estudiar a fons el funcionament d'openFrameworks, quines funcionalitats aportava i com estaven implementades. Com s'ha pogut anar veient en el Capítol 9, s'han anat trobant entrebancs i problemàtiques que s'han hagut d'anar resolent, com ara el tipus de *wrap* a utilitzar per les textures dels *shaders*, la diferència entre l'origen de coordenades de 'Shadertoy' i openFrameworks, les diferències en els tipus de renderitzats, com utilitzar correctament la il · luminació dins l'escena, i també, pel que fa a l'*script* en Python, s'ha hagut d'aprendre el funcionament de l'API de 'Shadertoy', per poder parsejar correctament els JSON i poder construir els *shaders* i els fitxers de sortida.

Pel que fa al resultat final, s'ha obtingut una aplicació que compleix els requisits marcats en aquest projecte. S'ha obtingut una aplicació que permet fer projeccions sobre models 3D d'una manera molt senzilla, utilitzant la tecnologia dels *shaders*, que és una de les utilitzades pels professionals del *projection mapping*, tot i que tenir-ne coneixement no és un requisit, ja que l'usuari pot desconèixer per complet la tecnologia. El funcionament de l'aplicació és molt intuïtiu, tot i que requereix de connexió a Internet, l'usuari desconeix com l'aplicació es comunica

amb 'Shadertoy'.

La interfície d'usuari dissenyada, gràcies als *addons* d'openFrameworks, compleix els requisits d'una manera senzilla, sense demanar massa entrada de dades a l'usuari, minimitzant així el percentatge d'error. A més, l'aplicació li permet a l'usuari tenir el total control sobre el model, el pot moure i rotar utilitzant el teclat i el ratolí.

S'ha utilitzat la tecnologia *render to texture* per poder "pintar" els *shaders* a la finestra o "mapejats" sobre un model 3D. El procés de descàrrega de *shaders* és ràpid, tot i l'inconvenient que alguns són privats i no es podran utilitzar.

Com a inconvenient més gran a destacar, hi ha el requisit a nivell de *hardware* que necessita l'aplicació per ser executada, sobretot pel que fa a la targeta gràfica. Tot i això, els ordinadors d'avui dia poden executar-la perfectament, i, probablement, la majoria de VDJs aficionats o professional ja disposen d'aquesta tecnologia.

Amb tot això, i com és d'esperar en un projecte d'aquestes dimensions, com s'ha pogut veure en el Capítol 4, la planificació inicial s'ha vist alterada, obtenint una planificació final més voluminosa, però que s'ha pogut complir obtenint bons resultats.

Com a reflexió final, s'han obtingut els resultats esperats, tot i les complicacions i desviacions que ha anat tenint el projecte. S'ha aconseguit una aplicació que fins ara no n'hi havia cap que utilitzés les tecnologies 'Shadertoy' i openFrameworks juntes, si que se n'han vist que utilitzen només les funcionalitats d'openFrameworks per fer projeccions, però no afegint la tecnologia dels *shaders*.

També és evident que és una aplicació resultat d'un treball final de grau, i que es pot millorar i afegir-hi noves funcionalitats, fins i tot utilitzant-ne d'openFrameworks per continuar en la mateixa línia. Tot i això, el resultat final és l'esperat, una aplicació que permet utilitzar els shaders com a projecció per a models 3D, per tenir una previsualització d'un *projection mapping* abans de passar a la producció professional, o bé com a joguina per a VDJs aficionats.

Capítol 12

Treball futur

Com s'ha comentat a la reflexió final de les conclusions, aquest projecte és el resultat d'un treball final de grau, i que pot tenir ampliacions, millores o treballs futurs, a partir dels resultats obtinguts.

Una de les possibles millores a implementar seria la d'afegir la funcionalitat d'un projector, per tenir un resultat més pròxim al real. És a dir, implementar la matemàtica que detecta els límits del model i té en compte la perspectiva d'aquest. Ara mateix l'aplicació deixa a l'usuari la responsabilitat de definir correctament les coordenades de textura del model 3D, per tant una solució seria que l'usuari no hagués de tenir en compte com s'haurà de projectar, sinó que ho tingués en compte un "projector", utilitzant, per exemple, llibreries com OpenCV. Fins i tot, intentar encastar aplicacions que ja permetin realitzar aquesta funcionalitat, dins l'aplicació d'openFrameworks.

Una altra possibilitat, per revertir la responsabilitat que té l'usuari de definir les coordenades de textura, però sense haver d'implementar un "projector", seria separar per una banda la projecció i per l'altra el model 3D, és a dir, renderitzar-ho per separat. Un cop renderitzar per separat, es podria agafar la textura resultant del renderitzat del *shader*, i modelar-la en temps real, per exemple amb el ratolí, per sobre del model 3D. Aquesta funcionalitat afegeix una característica més al projecte, assemblant-se al funcionament d'algunes aplicacions de *projection mapping*, podent controlar més la perspectiva com un projector real, però de més difícil control i precisió. Per exemple, utilitzant quelcom similar a l'*addon* d'openFrameworks *ofxMtlMapping2D*, que permet afegir punts de modelat a una superfície. En aquest cas podria ser la textura resultant del renderitzat del *shader*.

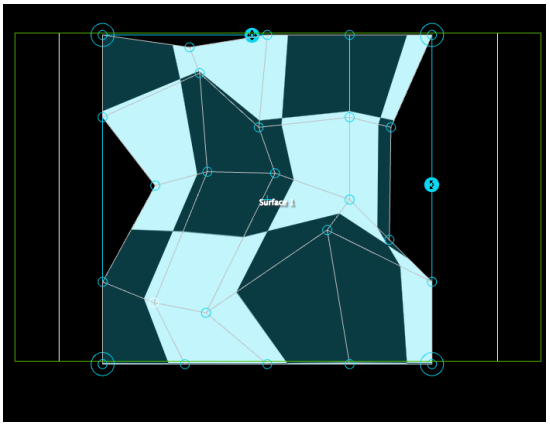


Figura 12.1: Modelat d'una superfície, a partir de definir-ne els punts, utilitzant *ofxMtlMapping2D*

Bibliografia

[1] jjtorres. (2013). *Hardware para novatos (III): ¿qué es y cómo funciona la GPU o tarjeta gráfica?*. Recuperat des de http://hipertextual.com/archivo/2013/12/hardware-gpu-grafica/

[2] Thiesen, M. (2010). *Video Graphics and Genomics: A Real Game Changer?*. Recuperat des de http://blog.goldenhelix.com/mthiesen/video-graphics-and-genomics-a-real-game-changer/

[3] Nvidia. (2003). *The Cg Tutorial*. Recuperat des de http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

[4] Shreiner, D., Sellers, G., Kessenich, J M., i Licea-Kane, B M. (2013). *OpenGL Programming Guide* (8th Edition). Reading, Massachusetts: Addison-Wesley Professional.

[5] Khronos Group. (2015). *OpenGL*. Recuperat des de https://www.opengl.org/

[6] García, M. (2010). *Programación de GLSL Shaders en OpenGL*. Recuperat des de http://www.abcdatos.com/tutoriales/tutorial/z9429.html

[7] Song Ho Ahn. (2014). *OpenGL Frame Buffer Object (FBO)*. Recuperat des de http://www.songho.ca/opengl/gl_fbo.html

[8] Jones, B., Sodhi, R., Karsch K., Casperson, M., Henke, C. (2015). *Projection Mapping Central*. Recuperat des de http://projection-mapping.org/

[9] Viquipèdia. (2014). *Vídeo Mapping*. Recuperat des de https://ca.wikipedia.org/wiki/Vídeo_mapping

[10] Python Software Foundation. (2015). *Python 3.4.3 documentation*. Recuperat des de https://docs.python.org/3.4/index.html

[11] The Code::Blocks team. (2015). *Code::Blocks*. Recuperat des de http://www.codeblocks.org/

[12] Lieberman, Z., Watson, T., Castro A. (2015). *openFrameworks*. Recuperat des de http://www.openframeworks.cc/

[13] Quilez, I., Jeremias, P. (2015). *Shadertoy*. Recuperat des de https://www.shadertoy.com/

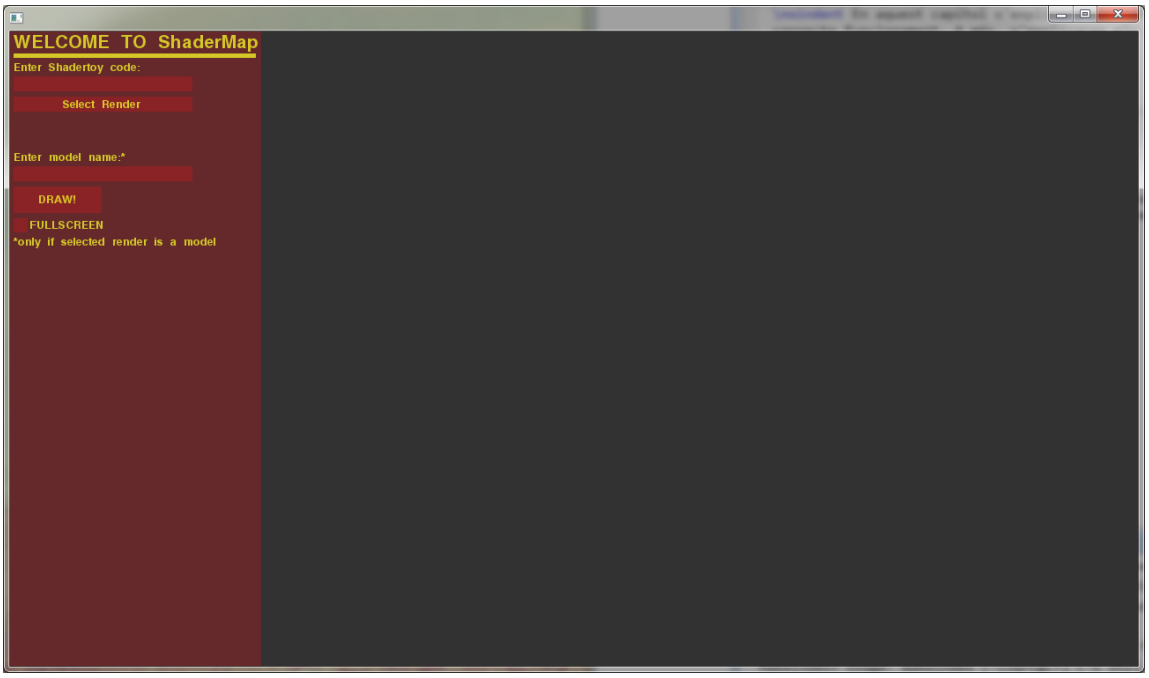
Capítol 13

Manual d'usuari i instal · lació

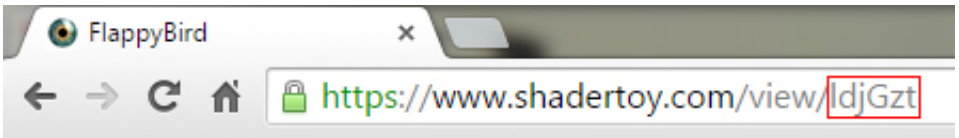
En aquest capítol s'explicarà com s'ha d'utilitzar l'aplicació per un correcte funcionament. A més, s'explicaran els passos a seguir per poder instal · lar-la.

13.1 Manual d'usuari

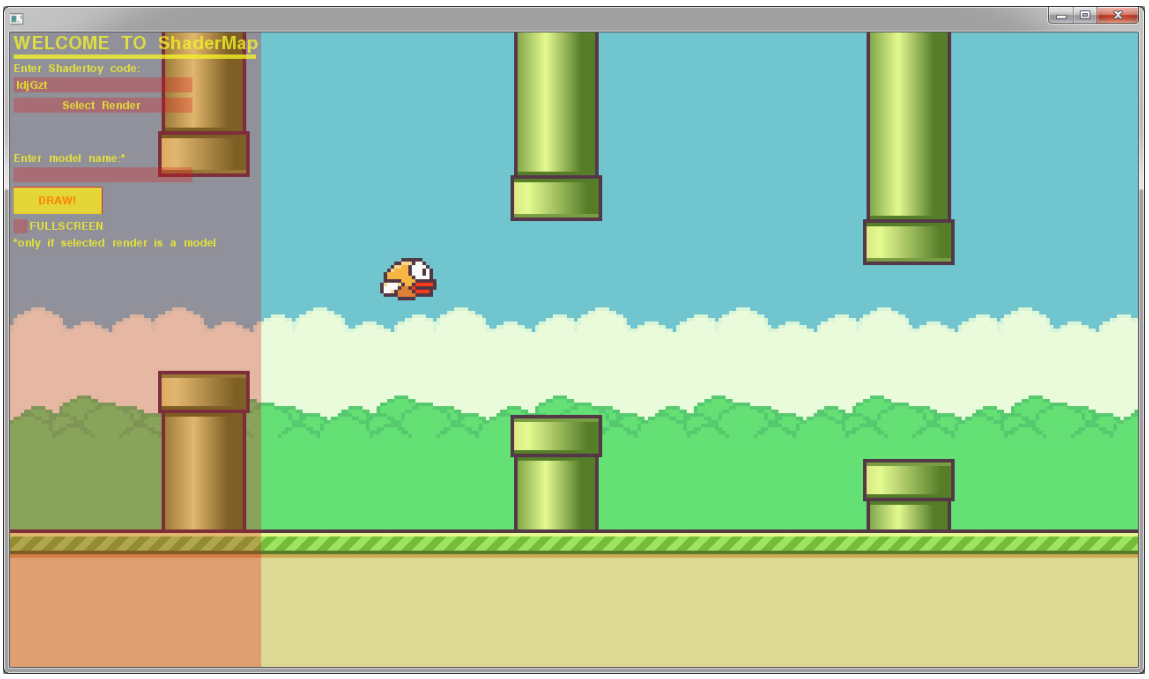
Podem començar executant l'aplicació, l'executable dins la carpeta "\bin" de l'aplicació. El primer que es veu quan s'executa l'aplicació és la interfície d'usuari amb la finestra buida:



Per començar a utilitzar l'aplicació, s'haurà d'escollir quin *shader* de 'Shadertoy' es vol utilitzar. Accedim a l'aplicació web i escollim un *shader*. S'haurà d'apuntar l'identificador del *shader*, com indica la següent imagte:

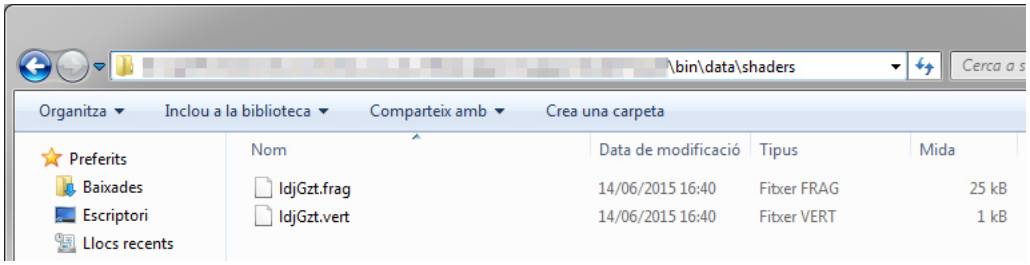


Aquest és el codi que s'ha d'entrar en el primer *textbox*. En aquest cas és: *ldjGzt*. Per tant, podem començar posant l'identificar i executar l'aplicació.

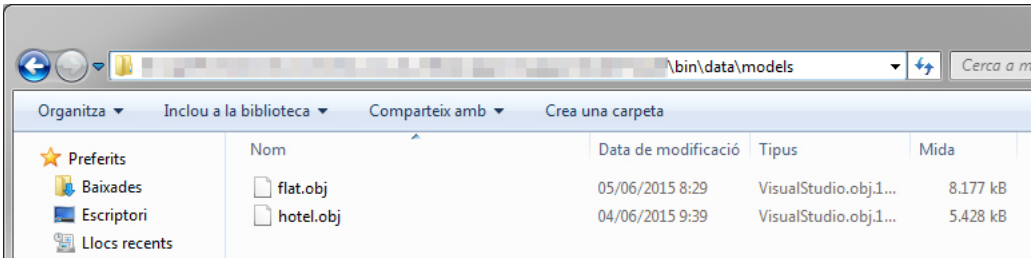


Si no es selecciona cap tipus de render abans de "pintar", per defecte es pinta a pantalla.

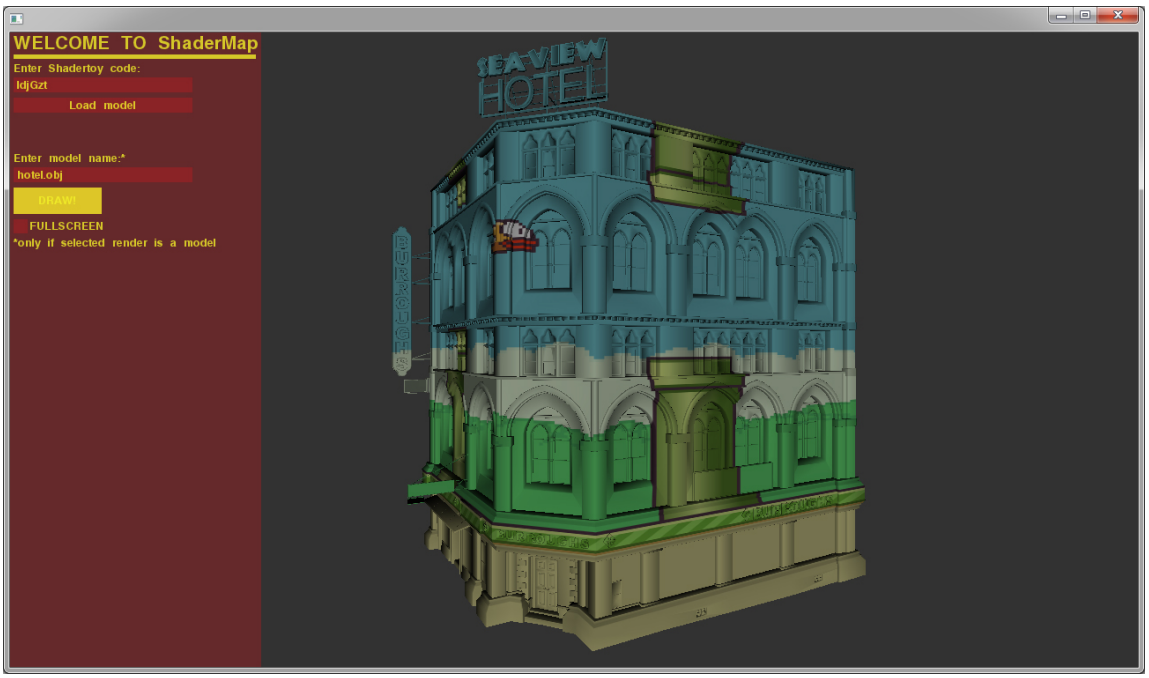
Els *shaders* que es vagin descarregant es guarden a la carpeta "\shaders" de l'aplicació.



A continuació, sense tancar l'aplicació en cap moment, podem seleccionar el tipus de render = "Load model" de la llista desplegable, i escriure el nom d'un model 3D, que haurem de tenir guardat a la carpeta "\models" de l'aplicació.



En aquest cas escriurem el nom del model "hotel.obj".



Com es pot veure, s'ha "mapejat" el *shader* per la superfície del model 3D "hotel.obj". Un cop carregat es pot moure i rotar lliurament utilitzant el teclat i el ratolí, i també canviar la vista a pantalla completa. A més, sense necessitat de tancar l'aplicació es pot escriure un nou *shader*, i un nou model si es prefereix.

Si ens equivoquem escrivint el nom del *shader*, o bé el nom del model 3D, o el *shader* no s'ha pogut descarregar perquè és privat, o no tenim el model guardat correctament, l'aplicació ho controla avisant-nos des de l'interpret de comandes. L'aplicació no s'atura i es poden tornar a escriure els paràmetres correctament.

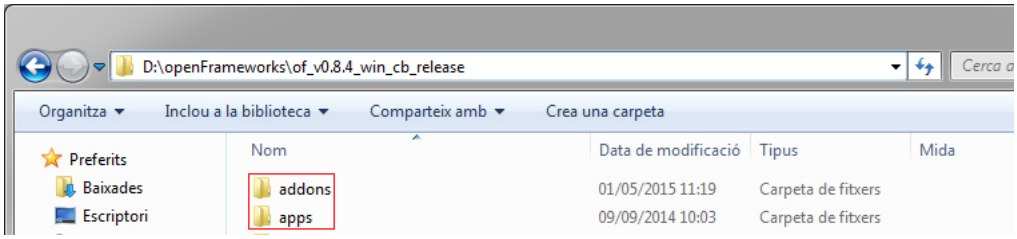
```
Shader ID is incorrect!
Shader ldjGzt downloaded successfully!

Model 'hotel.ob' doesn't exists!
```

En l'exemple anterior, primer s'ha escrit incorrectament el nom del *shader* i se'ns ha avisat. A continuació, s'ha rectificat i s'ha descarregat bé, però el model que havíem indicat per "mapejar" també s'havia escrit malament, i se'ns ha tornat a avisat.

13.2 Instal · lació de l'aplicació

El primer que s'ha d'instal · lar per utilitzar l'aplicació és l'openFrameworks: http://www.openframeworks.cc/versions/v0.8.4/of_v0.8.4_win_cb_release.zip

Un cop descarregat, s'hauran de tenir en compte dues carpetes dins del framework: "\addons" (carpeta on s'hauran d'instal · lar els *addons* utilitzats) i "\apps" (carpeta on s'instal · larà l'aplicació).



A continuació descarreguem els 3 *addons* utilitzats per aquesta aplicació:

- ofxCubeMap: https://github.com/andreasmuller/ofxCubeMap
- ofxUI: https://github.com/rezaali/ofxUI
- ofxAssimpModelLoader: Està incorporat en la versió descarregada de l'openFrameworks, no és necessari descarregar-lo.

Un cop descarregats, arrosseguem les carpetes de l'*ofxCubeMap* i l'*ofxUI* a la carpeta "\addons". Seguidament, arrosseguem la carpeta de l'aplicació dins la carpeta "\apps".

Finalment només queda guardar els models 3D que s'utilitzaran a la carpeta "\bin\data\models", com s'ha vist en el manual d'usuari.

Per continuar desenvolupant l'aplicació serà necessari instal · lar l'IDE amb el compilador, en aquest cas *Code::Blocks* amb *MinGW*. Per instal · lar-ho es poden seguir els passos de la guia d'openFrameworks: http://openframeworks.cc/setup/codeblocks/.