EPS
Escola Politècnica Superior
UdG

Projecte/Treball Fi de Carrera

Estudi: Eng. Tècn. Informàtica de Gestió. Pla 2001

Títol:

Entorn de treball per a visualitzar escenes tridimensionals

Document:

Memòria

Alumne: Salvi Pi Bonany

Director/Tutor: Mateu Sbert Casasayas

Departament: Informàtica i Matemàtica Aplicada

Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes/any): 01/08

índex

prefaci ... 3
introducció ... 4
objectius ... 5
anàlisi de casos d'ús ... 8
diagrames de casos d'ús ... 8
consideracions de disseny ... 11
wxWidgets ... 12
OpenGL, GLSL i Cg ... 14
OpenGL ... 14
exemple d'ús bàsic d'OpenGL ... 15
extensions OpenGL ... 16
biblioteca GLEW ... 16
GLSL ... 17
Cg ... 19
COLLADA ... 21
disseny de l'aplicació ... 23
diagrama de paquets ... 23
explicació del diagrama ... 24
diagrama de classes ... 25
explicació del diagrama ... 26
ButtonID (GUI) ... 26
ColladaLoader (Collada) ... 26
GUI_Canvas (GUI) ... 27
GUI_Dialog_SceneElement (GUI) ... 28
GUI_Frame_CanvasPanel (GUI) ... 28
GUI_Frame_MenuBar (GUI) ... 28
GUI_Frame_ScenePanel (GUI) ... 29
GUI_Frame_Settings_Camera (GUI) ... 29
GUI_Frame_Settings_Geometry (GUI) ... 30
GUI_Frame_Settings_Light (GUI) ... 31
GUI_Frame_Settings_Material (GUI) ... 32
GUI_Frame_Settings_Shaders (GUI) ... 32
GUI_Frame_Settings_Texture (GUI) ... 33
GUI_Frame_SettingsPanel (GUI) ... 33
GUI_Frame_SplitterWindow (GUI) ... 34
GUI_Frame_ToolBar (GUI) ... 34

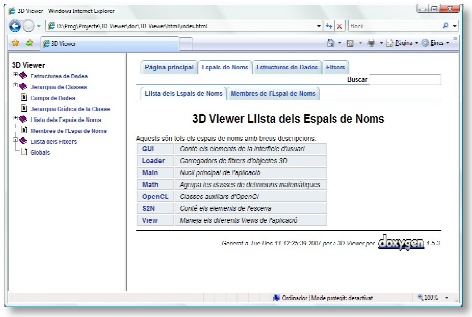
GUI_Frame_ViewsPanel (GUI) 34
GUI_MainFrame (GUI) 35
GUI_MainFrame_Identifiers (GUI) 36
LogicController (Main) 36
S2N_Cam (S2N) 38
S2N_OrthoCam (S2N) 39
S2N_PerspCam (S2N) 39
S2N_Geom (S2N) 40
S2N_Light (S2N) 41
S2N_Material (S2N) 42
S2N_Scene (S2N) 43
S2N_Texture (S2N) 44
S2N_Texture2D (S2N) 44
View_Elements (View) 44
View_Factory (View) 45
View_Interface (View) 45
Basic_FBO_View (View) 45
GLSL_View (View) 46
Lights_FBO_View (View) 46
LightsANDTexs_FBO_View (View) 46
ViewsHandler (Main) 47
ViewsWizard (Main) 48
ViewsWizard_CheckPage (Main) 48
especificació de les classes i els espais de noms 49
resultats 50
frame buffer 50
frame buffer amb llums 51
frame buffer amb llums i textura 52
GL Shading Language 53
Brick GLSL 55
conclusions 57
treball futur 59
bibliografia 60

annex A: Referència de Classes i Fitxers de Codi Font
annex B: Guia d'Usuari
annex C: Guia del Programador de *Views*

prefaci

El projecte ha generat diferents documents, el principal dels quals és aquesta memòria, i els seus annexes, així com l'aplicació en si i els seus arxius associats.

A la **memòria** ofereix una introducció a la matèria que tracte aquest projecte final de carrera, així com les motivacions i els objectius que aquest es proposava. A continuació es fa un somer repàs dels principals casos d'ús del sistema pretès, és a dir, quines funcions principals hauria d'acomplir el resultat del projecte per tal d'arribar a bon terme. Tot seguit entrem a considerar les principals eines i tecnologies emprades per desenvolupar l'aplicació resultant, amb unes petites definicions d'aquestes. Finalment es presenten alguns resultats de prova de l'aplicació, així com algunes consideracions sobre el treball futur que es podria realitzar.



L'**Annex A** de la memòria **Referència de Classes i Fitxers de Codi Font**, s'ofereix només en format digital pel seu volum (més de 800 pàgines) i es troba disponible a part de en format Word, en format ***html*** i ***chm.***

L'**Annex B** inclou una **Guia d'Usuari** per mostrar les funcionalitats principals de l'aplicació resultant així com la seva interfície.

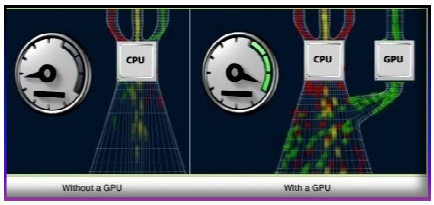
L'**Annex C** conté la **Guia del Programador de *Views***, que intenta presentar els passos que s'han de fer per programar un nou mòdul de visualització per a l'aplicació. Amb aquesta opció es pot estendre la funcionalitat del programa incorporant nous mètodes de visualització.

Pel que fa a la documentació inclosa al CD, s'inclouen les següents carpetes:

- **doc**, carpeta que inclou la documentació del projecte en format digital
    - **pdf**, documents en format PDF
    - **word97,** documents en format Word 97
    - **word07**, documents en format Word 2007, el nadiu a l'hora de realitzar la documentació
- **install**, consta d'un instal·lador executable per a sistemes Windows **Install.exe**, que conté els programes d'instal·lació de l'aplicació i el Codi Font
    - **bin,** només l'executable i les biblioteques necessàries
- **codi**, carpeta amb el codi font del projecte

introducció

El camp dels gràfics en tres dimensions és cada vegada més interessant i potent. Des de la irrupció de les computadores a les nostres vides, el salt qualitatiu que les ha acostat al nostre treball habitual ha estat la possibilitat d'usar interfícies gràfiques agradables i intuïtives. Aquells primers gràfics en dues dimensions, aviat van dirigir-se a pensar com es podrien aconseguir representacions d'objectes en tres dimensions, d'objectes reals. Així neix el concepte de model en tres dimensions, que especifica una sèrie de punts a l'espai, que defineixen una geometria concreta. És l'ordinador llavors l'encarregat de presentar-nos aquella figura des del punt de vista desitjat. Evidentment, si variem el punt de vista des d'on ens mirem aquell model, variarà també la presentació que veurem. Al principi aquest pas (veure la representació de l'objecte, que en diem ***renderitzat***), tan sols era possible com a resultat d'un costós procés de càlcul.



Amb l'imparable avanç de la informàtica, i sobretot de la informàtica gràfica, els ordinadors es dotaren de una circuiteria dedicada a processar elements gràfics cada vegada més potent. Avui dia parlem de **GPU**s (*Graphics Processing Unit*) programables, que ens permeten descarregar de feina la **CPU** (*Central Processing Unit*) de la computadora a l'hora de fer operacions amb gràfics. Diem que són programables, per què és possible encarregar part de la "feina" de visualització a aquests components. Així, amb tots aquests avanços, veiem en el nostre dia a dia com els models en tres dimensions poden ser visualitzats en el que anomenem **temps real**. Això significa que és possible fer els càlculs per visualitzar un model de manera tan ràpida que se'ns permet, per exemple, moure'ns al voltant d'un model en 3D i veure instantàniament el resultat.

Un dels grans impulsors del desenvolupament accelerat de la industria dels gràfics per computador han estat precisament les màquines de joc personals, tals com les famoses **Sony PlayStation** (http://es.playstation.com/), **Microsoft XBOX** (http://www.xbox.com/) o **Nintendo Wii** (http://wii.com/). Companyies punteres en aquest sector són **NVIDIA** (http://www.nvidia.com/) i **ATI** (http://ati.amd.com/), que generen les targetes de gràfics més potents del mercat personal.

S'ha d'esmentar també que es poden trobar més funcionalitats que les purament gràfiques per usar les targetes gràfiques d'última generació. Aquestes poden ser usades amb èxit, per exemple, per realitzar càlculs matemàtics amb requisits de paral·lelisme. El truc és representar les dades en forma de textures OpenGL i utilitzar els *shaders* (petits programes que rauen a la targeta gràfica i que reemplacen la funcionalitat fixa d'aquesta a l'hora de pintar per una altre al gust del programador) com a processadors que computen aquestes dades, que moltes vegades ho poden fer de manera paral·lela.

La introducció no estaria completa si no s'esmentés els estàndards **OpenGL** i **DirectX**. **OpenGL** és una especificació estàndard que defineix una API (Application Programming Interface, conjunt de definicions de funcions per als programadors) multillenguatge i multiplataforma per a escriure aplicacions que produeixen gràfics 3D. Desenvolupada originalment per **Silicon Graphics Incorporated** (http://www.sgi.com). OpenGL significa Open Graphics Library, que traduït és "biblioteca de gràfics oberta". **Microsoft DirectX** és també una API per realitzar les tasques relatives a temes multimèdia, més específicament a la programació de jocs i vídeo. Es troba majoritàriament sobre plataformes Microsoft (XBoX, sistemes operatius Windows) i permet l'explotació d'altes prestacions de les capacitats gràfiques d'un ordinador. La majoria de targes gràfiques d'altes prestacions actuals incorporen les funcions d'aquests dos estàndards i les acceleracions hardware que aquestes implementen per aquestes funcions.

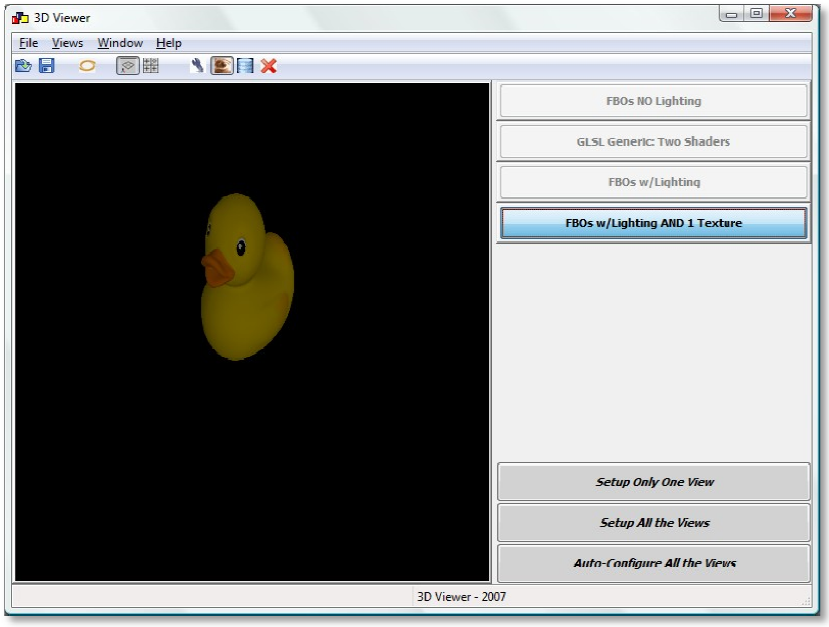
Per veure una explicació més detallada dels conceptes que apareixen en aquesta introducció podem referir-nos a l'apartat **consideracions de disseny** d'aquest document per una explicació més extensa.

objectius

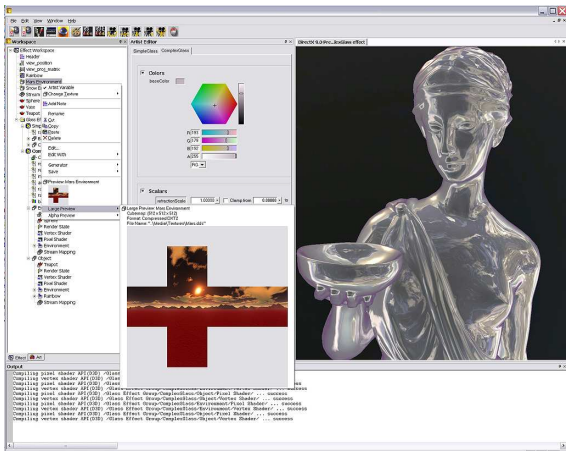
La intenció d'aquest projecte és proporcionar un entorn de treball per tal de visualitzar models tridimensionals en temps real. Aquest projecte perdria part del seu sentit si només pretengués això, ja que existeixen, tan al mercat com en codi lliure, molts programes que realitzarien aquesta tasca a la perfecció. Repassem però els objectius que es marcà aquest projecte al iniciar-se:

a) Proporcionar una interfície gràfica per poder visualitzar interactivament una escena, modificant-ne els seus elements, com llums, càmeres i mètodes de visualització entre altres.

b) Aconseguir un disseny que faci el projecte altament revisable i reutilitzable en el futur, i serveixi per tant de plataforma per provar altres projectes, com per exemple diferents tipus de càlculs d'il·luminació.



Pel que fa al primer objectiu, el d'aconseguir una interfície gràfica per visualitzar l'escena, que fos amigable i entenedora es podria dir que s'ha assolit amb força èxit. S'han usat per aconseguir-ho directrius que han donat una presència semblant a la que tenen la majoria de programes sota **Windows.** Presentant elements que se'ns han fet tan comuns, com menús, barres d'eines, etc. s'ha buscat facilitar-ne la operativitat i maneig. S'ha usat per això la biblioteca **wxWidgets**, que té la propietat d'adaptar-se al *feel & look* del sistema (podem veure a la imatge de la pàgina anterior el *feel & look* de Windows Vista per exemple).



Ara bé, és el segon objectiu el que diferencia el projecte de qualsevol visor de models tridimensionals. Es pot dir que aquest objectiu és bastant relatiu al que s'entengui per revisable i reutilitzable, però s'ha intentat estructurar el codi i proporcionar facilitats per aconseguir que ho sigui.

Hi ha al mercat altres eines que acompleixen millor els objectius de crear nous models de visualització, ja sigui usant *shaders*, com per exemple el **RenderMonkey (**desenvolupat per **ATI)** o o bé d'altres tècniques. Encara que aquest últim permet treballar amb els *shaders* de forma molt intuïtiva i amb infinitat d'opcions, ens amaga l'apartat d'ordres d'OpenGL. És a dir, les ordres prèvies a l'execució del *shader*, tals com decisions d'usar **Frame Buffer Objects** (objectes especials d'OpenGL que ens permeten renderitzar offline), o bé pintar usant **CallLists** (conjunt d'ordres per passar primitives a OpenGL), o bé usant **DrawElements** (una altra funció d'OpenGL que ens permet "carregar" els vèrtexs a la targeta gràfica i accelerar-ne el pintat) , o bé simplement efectuar càlculs previs per obtenir el valor d'un paràmetre, que passarem després a un determinat *shader*. Per tant, des de una vessant educacional i pedagògica, el projecte presentat ens permet seguir tot el cicle de creació de la imatge, des de les ordres OpenGL a les ordres dels *shaders*, cosa que en el programa referit anteriorment no és tan fàcil. Amb l'aplicació que s'ha volgut desenvolupar s'ha preferit preservar la llibertat de programació d'OpenGL per no coartar-ne cap vessant.

Aquest segon objectiu ve al cas per què, com ja he comentat anteriorment, la informàtica gràfica avança amb passes de gegant, i és molt important poder disposar d'una base, d'un codi provat, que pugui incloure els nous desenvolupaments i estàndards de la industria. Al llarg del desenvolupament d'aquest projecte mateix, els estàndards d'**OpenGL** s'han anat succeint de versió en versió, en un lapse força curt de temps.

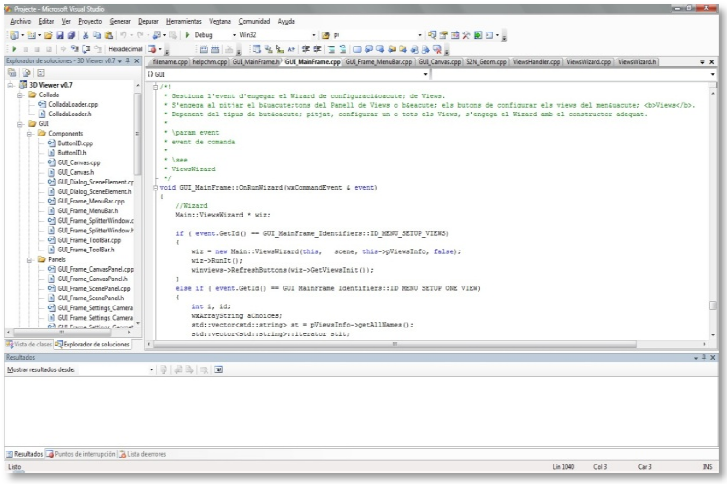
Al començament del projecte, per exemple, es va investigar la possibilitat d'usar el llenguatge de programació de les targes gràfiques d'alta gamma **Cg** (http://developer.nvidia.com/page/cg_main.html), desenvolupat per **NVIDIA**. Finalment amb el propi avanç d'**OpenGL** i la incorporació del seu propi llenguatge **GLSL** per programar targes gràfiques, es va optar per aquest últim. Amb això només vull ressenyar el ritme vertiginós (i les conseqüents dificultats d'implementació) d'aquest sector. És per això que s'ha intentat aconseguir un sistema que faciliti la prova de les noves funcionalitats de les targetes gràfiques.

A la finalització d'aquest projecte, per mostrar un altre exemple del que es deia anteriorment, havia aparegut la definició d'OpenGL dels **Geometry Shaders**, que són un tipus especial de *shaders* que són capaços de generar nous vèrtexs usant com a entrada les primitives d'OpenGL (les primitives són les ordres que defineixen objectes senzills, com punts, línies, triangles, ...).

Per exemple, una sèrie de punts passats com a primitives al *pipeline* d'OpenGL, després de passar per un programa de *geometry shader*, poden esdevenir tota una altra sèrie de punts, que seran utilitzats per dibuixar.

Aquest objectiu d'aconseguir aquesta llibertat de programació s'ha estructurat en el que a la documentació s'anomena *View*. Així un *View* és el mòdul de codi encarregat de definir quina tècnica volem utilitzar per pintar els elements d'una escena. Un *View* per exemple pot només usar ordres bàsiques de OpenGL per pintar la geometria, sense tenir en compte la il·luminació. També se'n poden definir de més complexos, que incloguin la il·luminació, les textures, el pas per un o varis *shaders*, ... Aquest aspecte queda a l'entera disposició que en faci el programador de *Views*, que serà qui definirà quins elements s'han de requerir per usar aquell *View*, i quines tècniques usarem per pintar-los. L'únic requisit és que al final retornem una textura que pugui ser pintada per l'aplicació.

Es pot objectar, amb força encert, que el sistema resultant no és ni molt menys interactiu i senzill, ja que requereix recompilacions del codi font i coneixements de programació. Tot i això, s'ha descartat la possibilitat d'usar llenguatges interpretats (tipus Java o Python) pel cost en velocitat que això té, sense contar que el llenguatge de programació **C++,** l'escollit, és el que més aviat ens acosta a les noves funcionalitats tan lligades al hardware. També s'ha descartat crear una parametrització estricte de l'estat d'OpenGL per modificar-lo des de l'aplicació, per l'abast i els canvis que aquest va sofrint. Finalment, el procés de prova de les noves funcionalitats dels estàndards o les targetes d'última generació requereixen coneixements d'usuari avançat i per tant no considero que tot això sigui una limitació considerable.

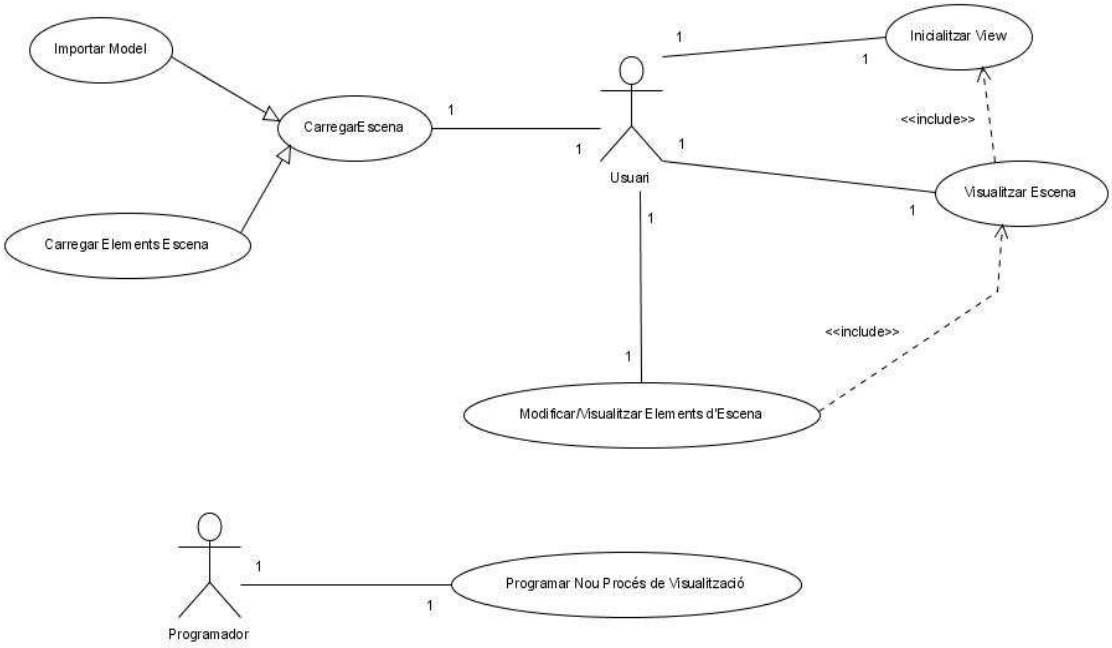


El **resultat final** del desenvolupament del projecte és per tant un entorn de visualització de models tridimensionals, que en aquest primer estadi, han de ser models triangulats de l'estàndard COLLADA. El programa resultant s'ha anomenat ***3D Viewer***, i, com s'ha esmentat al prefaci, se n'inclouen els installadors i el codi font al CD acompanyant. Aquesta aplicació ens presentarà les diferents opcions de visualització disponibles i les eines per visualitzar els elements de l'escena. Per un millor enteniment de l'ús i la funcionalitat d'aquesta aplicació, referir-se sisplau a l'**annex B** d'aquesta memòria, la **Guia d'Usuari**, també disponible al CD acompanyant.

anàlisi de casos d'ús

Per tal d'aconseguir els objectius marcats, fem un cop d'ull, primer de manera esquemàtica als principals casos d'ús de l'aplicació que es vol desenvolupar. Els diagrames que es mostren a continuació (basats en UML, Unified Modelling Language, http://www.uml.org), no pretenen ser representacions exhaustives en si mateixes, i per això s'intentarà usar els diagrames només com a complement i ajuda a l'explicació.

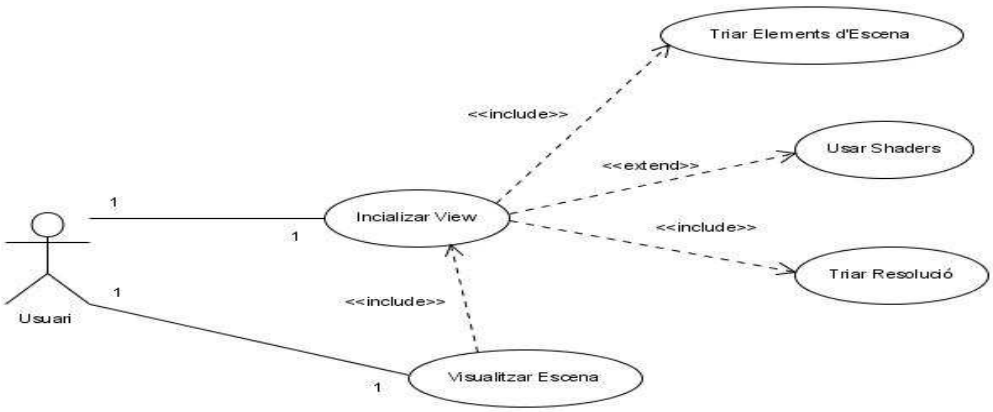
diagrames de casos d'ús



El diagrama de context en mostra dos casos completament separats d'ús. Per una banda hi ha l'ús normal que qualsevol **usuari** pot esperar de l'aplicació per mostrar un model en tres dimensions, i per l'altra el **programador** que investiga noves funcionalitats. Són àmbits molt diferents, i aquí em centraré en el que concerneix a la pròpia aplicació. Un programador de nous processos de visualització s'haurà d'enfrontar, depenent de la tècnica o funcionalitat que vulgui implementar a un conjunt de variables sempre canviant, que no es podria, ni és l'objectiu, reproduir en aquest document.

Així doncs centrem-nos en l'**usuari** de l'aplicació. Les funcions principals que voldrà fer qualsevol usuari en aquest tipus d'aplicació, és **a**.-carregar l'escena (models 3D, llums, càmeres, etc.), **b.-**modificar els elements d'aquesta i **c**.-visualitzar l'escena en si.

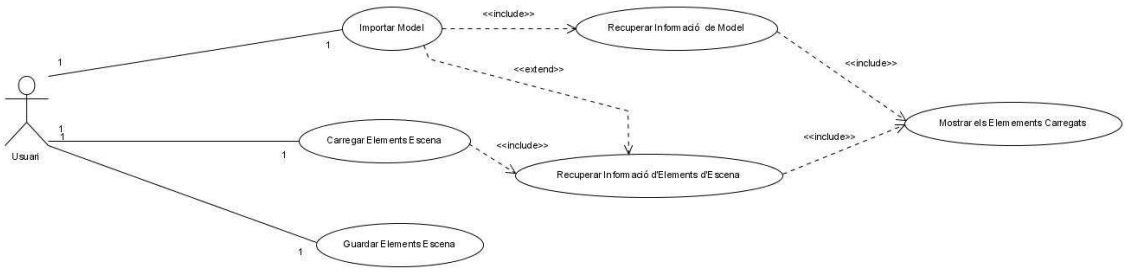
a) Per tal de **visualitzar l'escena**, haurem de triar el tipus de visualització que volem usar d'entre les disponibles. A partir d'ara anomenarem a aquests modes de visualització diferents ***Views***. Un *View* és el mòdul responsable de prendre els elements de l'escena necessaris i visualitzar-los d'acord amb el seu codi. Un *View* és per tant allò que un programador haurà de desenvolupar per tal d'integrar a l'aplicació un nou mode de visualització. Per tal de visualitzar l'escena usant alguns d'aquests *Views*, l'usuari haurà d'inicialitzar-lo, tot indicant-li quins elements de l'escena vol usar. Tal com vol reflexar l'esquema, per tal visualitzar una escena amb un cert *View*, s'haurà d'haver inicialitzat abans aquest *View*, és clar. Examinem-ho un xic més:



Tal com hem dit abans, per visualitzar una escena, cal que algun *View* estigui primer inicialitzat correctament. Per inicialitzar un *View*, haurem de triar els elements de l'escena que volem usar d'entre els que necessita aquell *View*. Si el *View* necessita **shaders** (petits programes que s'instal·len a les **GPU**s de les targes gràfiques), els podrem definir aquí també, així com triar la resolució del renderitzat resultant.

b) Entenem l'acció de **carregar elements de l'escena**, tan el fet de carregar un model en tres dimensions, com el fet de carregar diferents elements de l'escena, tals com llums, càmeres, materials i textures. Això és el que representen les dos subclasses **Importar Model** i **Carregar Elements Escena**. El primer es refereix a importar un model definit en algun format conegut per l'aplicació i el segon a carregar els elements abans descrits.

Examinem una mica més detingudament el fet de manejar i **carregar elements de l'escena**. Tenim dos escenaris diferents: importar un model tridimensional i recuperar i guardar altres elements de l'escena.



Guardar elements d'escena no té cap secret, simplement s'examina l'escena carregada a l'aplicació i es desa en un fitxer amb el format adient.

Per **carregar els elements de l'escena**, examinem un fitxer d'escena i en recuperem la informació dels elements. Aquests elements recuperats passaran a formar part del conjunt d'elements de l'escena de l'aplicació i es mostraran a la llista d'elements de l'escena.

Finalment, a l'hora **d'importar un model**, en recuperarem la informació de la geometria i ho mostrarem com un altre element de l'escena. Hi ha la possibilitat que un model dugui informació sobre altres elements de l'escena, com llums, càmeres, materials, ... També recuperarem opcionalment aquests elements i els mostrarem.

c) A l'hora de **modificar els elements de l'escena**, ens referim a modificar la posició i característiques d'elements com les càmeres, els llums o els materials. Aquests canvis en els diferents elements de l'escena es reflexaran en la visualització de l'escena.

consideracions de disseny

A l'hora de dissenyar l'aplicació s'han hagut de superar una sèrie de problemes i necessitats, així com prendre les decisions de l'entorn i les eines de programació a utilitzar.

Es pretenia que l'aplicació utilitzés el potencial de les targes gràfiques d'última generació, que tenen, com s'ha explicat capacitat de programació. Per desenvolupar el projecte per tant, el Grup de Gràfics de I.M.A., em va cedir amablement una estació de treball ben dotada per realitzar el treball. La decisió sobre la plataforma de desenvolupament va estar aviat clara. Al Grup, usaven majoritàriament el Microsoft Visual C++ per desenvolupar les seves aplicacions. Es va adoptar doncs aquesta mateixa plataforma que ja estava instal·lada i amb les llicències necessàries per realitzar la implementació de l'aplicació. No obstant, es volia conservar una certa independència de l'aplicació respecte el sistema operatiu, així que es van estudiar diversos **toolkits**. Un **toolkit** és un conjunt de controls gràfics i les seves operacions i gestions associades, que s'utilitza per bastir una interfície gràfica.

En un primer moment es va pensar en usar **Qt**, un toolkit propietari desenvolupat per **Trolltech** (http://www.trolltech.com), que era multiplataforma i amb moltes aplicacions i opcions. En aquell moment però es va desestimar per l'elevat cost d'una llicència sota Windows que suposava.

Es va examinar també la **Fox Toolkit** (http://www.fox-toolkit.org) i de fet les primeres proves i versions de l'aplicació es van realitzar amb aquest conjunt d'eines. Aquest *toolkit* està llicenciat sota la llicència lliure **GPL**, i és també multiplataforma. Es va acabar descartant per problemes per integrar les ordres OpenGL dins el *toolkit* i la seva llicència restrictiva.

Finalment es va optar pel *toolkit* **wxWidgets** (http://www.wxwidgets.org), un *toolkit* amb una llicència més relaxada que la GPL però que permet construir sistemes multiplataforma. A més posseeix una àmplia i bona documentació, a més de solvència i experiència contrastada.

Així va quedar establert el sistema operatiu (**Windows**), l'entorn de desenvolupament (**Visual Studio 2005**) i el llenguatge de programació (**C++**) i el toolkit a utilitzar per guanyar independència sobre el sistema operatiu (**wxWidgets**).

A continuació s'ofereixen unes breus descripcions d'algunes les eines usades per construir l'aplicació, de nom ***3D Viewer,*** que funcionarà en un entorn de **PC Windows**, i que en **resum** es podrien resumir en:

- **Microsoft Visual Studio 2005**, com a entorn de desenvolupament en C++.
- **wxWidgets**, la biblioteca per crear la interfície gràfica i altres funcions accessòries.
- **GLEW**, la biblioteca per inicialitzar i usar extensions d'OpenGL.
- **OpenGL**, per les instruccions de dibuix dels *Views* i també de certes operacions dels elements de l'escena. També s'utilitzen les extensions OpenGL adients per crear certs efectes.
- **COLLADA**, tan l'esquema en si per els models tridimensionals, com les biblioteques per accedir-hi i llegir les geometries.

wxWidgets

Pàgina Web: http://www.wxwidgets.org/

Les wxWidgets són unes biblioteques multiplataforma i lliures, usades pel desenvolupament de interfícies gràfiques programades en llenguatge C++. Estan publicades sota llicència LGPL, similar a la GPL, amb la excepció que el codi binari produït, pot ser propietari, permetent desenvolupar aplicacions comercials sense cost.

Les wxWidgets proporcionen una interfície gràfica basada en les biblioteques ja existents al sistema operatiu (anomenades **natives**), així que s'integra de forma òptima a aquest i resulten molt portables entre diferents sistemes operatius. Estan disponibles per Windows, MacOS, GTK+, Motif, OpenVMS i OS/2.

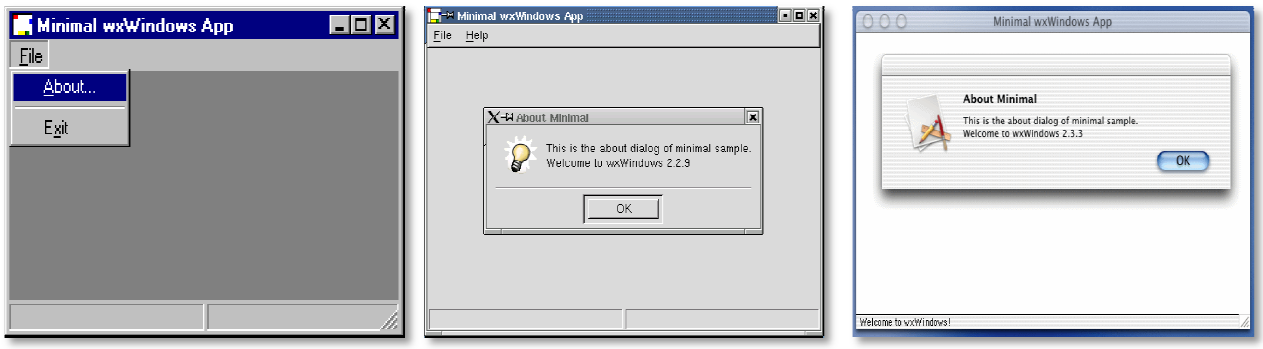
A més, també poden ser utilitzades des d'altres llenguatges de programació, a part del C++, tals com Java, Javascript, Perl, Python, Smalltalk i Ruby .

La última versió és la 2.8.6 d'Octubre 2007, que és la que s'ha acabat integrant en el desenvolupament de l'aplicació.

wxWidgets foren dissenyades el 1992 a l'Institut d'Aplicacions d'Intel·ligència Artificial de la Universitat de Edimburg per Julian Smart. Julian dissenyava una eina meta-CASE (anomenada Hardy) que necessitava córrer sota Windows, així com en estacions de treball de X-Unix. Les eines existents i comercials multiplataforma eren massa costoses per un projecte experimental, així que la seva única alternativa era crear la seva pròpia eina: **wxWidgets** (w de **Windows** i la x per les **X** de UNIX), que va començar suportant Xview, MFC 1.0, i AIAI.

Inicialment s'anomenaven **wxWindows** però va haver de canviar-se a **wxWidgets** perquè Microsoft va posar una demanda a finals del 2003 perquè podria existir confusió amb el seu Sistema Operatiu.

És un entorn semblant a **MFC** (Microsoft Foundation Class, conjunt de classes en C++ per accedir a l'API de Windows i als seus objectes com botons, barres, ...) especialitzat en el desenvolupament d'aplicacions multiplataforma en llenguatge C++, encara que també existeixen, com s'ha explicat, implementacions en altres llenguatges. És multiplataforma i suporta Windows, Linux, Mac OS X, Unix i les seves variants Solaris, així com plataformes mòbils com Microsoft Pocket PC i Palm OS.



Es distribueix sota llicència Open Source i GNU LGPL (Lesser General Public License) permetent utilitzar-la per desenvolupaments comercials.

Està composada per una part denominada **wxBase** que inclou classes com **wxString**, classes per gestionar arxius i directoris de manera independent del sistema i funcionalitats com: gràfics 2D, 3D amb **OpenGL**, bases de dades (ODBC), xarxes, impressió, fils d'execució, visió i impressió de HTML i un sistema d'arxius virtual.

Les raons per escollir **wxWidgets** són, a més de les seves característiques, que compta amb suport, documentació a Internet, ajuda en línia, fòrums, tutorials en diversos formats i desenvolupadors a la xarxa que hi tenen interès i hi veuen futur, un sistema flexible a events, crides a gràfic, etc. A més suporta **MDI** (Multiple Document Interface) i es poden crear DLLs sobre Windows i biblioteques dinàmiques en Unix.

OpenGL, GLSL i Cg

OpenGL

Pàgina Web: http://www.opengl.org

OpenGL és una especificació estàndard de la indústria que defineix una API multillenguatge i multiplataforma per a escriure aplicacions que produeixen gràfics 2D i 3D. La interfície consisteix en més de 250 crides, que poden ser usades per dibuixar complexes escenes tridimensionals usant simples primitives (operacions senzilles de baix cost computacional). Desenvolupada originalment per Silicon Graphics Incorporated (SGI) el 1992, és àmpliament usada en CAD, realitat virtual, visualització científica, simulacions de vol, etc. També és usada en videojocs, on competeix amb Direct3D a les plataformes Microsoft Windows. OpenGL significa **Open G**raphics **L**ibrary, que traduït és "biblioteca de gràfics oberta".

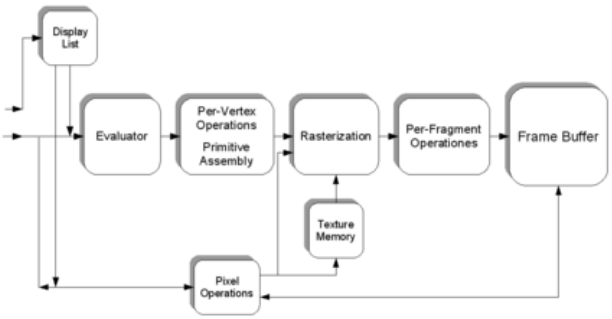
Entre les seves característiques podem destacar que és multiplataforma (hi ha fins i tot un OpenGL ES per a telèfons mòbils) i que la gestió de la generació de gràfics 2D i 3D per hardware ofereix al programador gràfic una API senzilla, estable i compacta. A més la seva escalabilitat ha permès que no s'hagi estancat el seu desenvolupament i també la creació d'extensions, una sèrie d'afegits sobre les funcionalitats bàsiques, per tal d'aprofitar les creixents evolucions tecnològiques. Podem ressenyar la inclusió dels GLSL (un llenguatge de *shaders* propi) com a estàndard en la versió 2.0 d'OpenGL. Al ser multiplataforma podem trobar OpenGL per a moltes plataformes (Linux, Unix, MacOS, Microsoft Windows, etc.). Però a Linux a més trobem la implementació lliure Mesa.

OpenGL serveix a dos objectius bàsics:

a) Amagar i encapsular les complexitats de manejar les diferents targetes acceleradores 3D, presentant al programador una única API uniforme.
b) Amagar les diferents capacitats de les plataformes hardware, requerint que totes les implementacions suportin tot el conjunt d'ordres OpenGL (usant emulació software si és necessari).

L'operativa bàsica d'OpenGL és la d'acceptar primitives tals com punts, línies i polígons, i convertir-los a **píxels** (mínima unitat d'informació representada o pintada a la pantalla d'un ordinador). Aquesta feina la fa el que anomenem *graphics pipeline* ("tuberia" encarregada de processar les primitives), coneguda com la màquina d'estats d'OpenGL. Moltes de les comandes d'OpenGL consisteixen en llançar primitives a la *graphics pipeline*, o bé configurar com la *pipeline* processa aquestes primitives. Al configurar la *pipeline*, el que fem és variar una màquina d'estats

OpenGL és una API de baix nivell, que requereix que el programador dicti pas per pas tot allò requerit per dibuixar una escena. Això és força diferents d'altres llibreries gràfiques, que només requereixen una descripció de l'escena i que deixen la responsabilitat del renderitzat a la pròpia llibreria. El disseny de baix nivell d'OpenGL requereix que els programadors tinguin un bon coneixement de la *graphics pipeline*, però això també dóna una gran llibertat per implementar noves maneres de renderitzar. És per aquest últim motiu que s'ha escollit aquesta API per



desenvolupar l'aplicació, a part de ser multiplataforma: és possible treballar amb les últimes novetats del hardware i investigar noves representacions.

Una descripció senzilla del procés de la *pipeline* d'OpenGL seria, com representa el gràfic:

1. Especificar la **geometria** a dibuixar a través de les primitives (punt, línia, triangle, quad, polígon).
2. **Avaluació** de les funcions polinòmiques, tals com les superfícies NURBS, per aproximar-les a les primitives adients.
3. **Operacions Per-Vèrtex**, transformant-los i il·luminant-los depenent del seu material. També descartant les parts no visibles de l'escena, per tal de no produir parts innecessàries de l'escena.
4. **Rasterització** o conversió de la informació anterior en píxels. Els polígons són representats amb el color apropiat usant algoritmes d'interpolació.
5. **Operacions Per-Fragment**, com ara refrescar alguns valors dels píxels depenent de l'entrada i dels valors prèviament emmagatzemats dels valors de profunditat, o bé realitzar combinacions de color, entre d'altres.
6. Finalment, els fragments generats són introduïts al ***Frame Buffer*** (lloc de la memòria del sistema on s'emmagatzemen els píxels que el sistema operatiu s'encarregarà de mostrar després per pantalla).

La majoria de les targetes acceleradores 3D proveeixen funcionalitats molt més avançades que aquesta *pipeline*, però aquestes noves característiques són generalment només millores, i no pas reinvencions radicals d'aquesta.

exemple d'ús bàsic d'OpenGL

```
glClear( GL_COLOR_BUFFER_BIT );
```

Això neteja el buffer de color.

```
glMatrixMode( GL_PROJECTION );      /* Subsequent matrix commands
will affect the projection matrix */
glLoadIdentity();                   /* Initialise the projection
matrix to identity */
glFrustum( -1, 1, -1, 1, 1, 1000 ); /* Apply a perspective-
projection matrix */
```

Això inicialitza la matriu de projecció, configurant una matriu que representa l'àrea visible. Aquesta matriu s'encarrega de transformar els objectes del punt de vista de la càmera a l'espai de projecció OpenGL.

```
glMatrixMode( GL_MODELVIEW );       /* Subsequent matrix commands
will affect the modelview matrix */
glLoadIdentity();                   /* Initialise the modelview to
identity */
glTranslatef( 0, 0, -3 );           /* Translate the modelview 3
units along the Z axis */
```

Aquestes comandes inicialitzen la matriu modelview. Aquesta matriu defineix una transformació des de coordenades relatives al model a les de l'espai de la càmera. La combinació de la matriu modelview amb la matriu de projecció transforma objectes des de l'espai relatiu al model fins l'espai de projecció a la pantalla.

```
glBegin( GL_POLYGON );              /* Begin issuing a polygon */
glColor3f( 0, 1, 0 );               /* Set the current color to
green */
glVertex3f( -1, -1, 0 );            /* Issue a vertex */
glVertex3f( -1, 1, 0 );             /* Issue a vertex */
glVertex3f( 1, 1, 0 );              /* Issue a vertex */
glVertex3f( 1, -1, 0 );             /* Issue a vertex */
glEnd();                            /* Finish issuing the polygon
*/
```

Aquestes comandes dibuixen un quadre verd al pla XY

extensions OpenGL

Pàgina Web: http://www.opengl.org/documentation/extensions/

L'estàndard OpenGL permet als fabricants proveir funcionalitats addicionals a través del que s'anomenen **extensions**. Les extensions poden introduir noves funcionalitats i noves constants i poden relaxar o obviar restriccions de les funcions OpenGL existents. Cada fabricant té assignat una abreviació que és usada per donar nom a les noves funcions o constants afegides per aquest fabricant. NVIDIA per exemple usa l'abreviació **NV**.

Quan el comitè **Architecture Review Board (ARB)** accepta una extensió per formar part de la pròxima versió d'OpenGL, aquesta rep l'abreviatura ARB, per exemple GL_ARB_multitexture. Per usar una extensió, abans haurem de determinar si està disponible per la nostra tarja gràfica, i després obtenir-ne els punters a les funcions. Les biblioteques com les **GLEW** o les **GLEE** estan pensades per facilitar aquests processos.

biblioteca GLEW

Pàgina Web: http://glew.sourceforge.net/

La Open**GL** **E**xtension **W**rangler Library (GLEW) és una biblioteca en C/C++ multiplataforma que ajuda en les consultes i la càrrega de les extensions OpenGL. GLEW aporta mecanismes per determinar quines extensions OpenGL estan suportades per la plataforma de treball. GLEW funciona per una gran varietat de sistemes operatius, incloent Windows, Linux, Mac OS X, FreeBSD, IRIX, i Solaris. GLEW està distribuïda sota la llicència BSD modificada.

A l'aplicació s'ha decidit incloure les GLEW de manera nativa, per ajudar a proporcionar les eines necessàries per programar mòduls capaços d'aprofitar tota la potència de les targes gràfiques. S'haurà de tenir en compte que s'hauran d'anar actualitzant els arxius de GLEW per aconseguir-ne les noves versions. S'ha decidit *linkar* la biblioteca de manera estàtica, per accelerar-ne els accessos.

GLSL

Pàgina Web: http://www.opengl.org/documentation/glsl/

Amb l'aparició de OpenGL 2.0 va aparèixer el llenguatge de programació de *shaders* a l'estil de **C** anomenat Open**GL S**hading **L**anguage, també conegut com **GLSL** o **slang**. El seu propòsit és reemplaçar la funcionalitat fixa dels processos de vèrtexs i fragments per petits codis, anomenats *shaders* que reemplacen aquesta funcionalitat fixa per una de nova. Amb la versió d'OpenGL 2.1, el GLSL inclou la seva versió 1.20. L'opció d'escriure *shaders* que poden ser usats en tots els fabricants que suportin OpenGL 2.0 és una característica molt interessant.

L'aplicació consta de la possibilitat d'afegir *shaders* als mòduls de visualització que així ho requereixin, ja que formen part de les noves funcionalitats de les targes gràfiques d'última generació.

Aquí hi ha un petit compendi dels detalls més rellevants de GLSL, per tal de fer-nos una idea del llenguatge:

Tipus de Dades

- **void** – functions que no retornen cap valor
- **bool** – condicional, vedader o fals
- **int** – enter amb signe
- **float** – nombre en coma flotant
- **vec2** – vector de 2 components en coma flotant
- **vec3** – vector de 3 components en coma flotant
- **vec4** – vector de 4 components en coma flotant
- **bvec2** – vector de 2 components booleans
- **bvec3** – vector de 3 components booleans
- **bvec4** – vector de 4 components booleans
- **ivec2** – vector de 2 components enters
- **ivec3** – vector de 3 components enters
- **ivec4** – vector de 4 components enters
- **mat2** – matriu de 2X2 amb nombres en coma flotant
- **mat3** – matriu de 3X3 amb nombres en coma flotant
- **mat4** – matriu de 4X4 amb nombres en coma flotant
- **sampler1D** – manejador per accedir a una textura d' 1 dimensió
- **sampler2D** – manejador per accedir a una textura de 2 dimensions
- **sampler3D** – manejador per accedir a una textura de 3 dimensions
- **samplerCube** – manejador per accedir a una textura mapejada en un cub
- **sampler1Dshadow** – manejador per accedir a una textura de profunditat de 1 dimensió
- **sampler2Dshadow** – manejador per accedir a una textura de profunditat de 2 dimensions

Operadors

A excepció dels operadors de bits i els dels punters, tots els altres operadors segueixen la forma de C/C++.

Funcions i estructures de control

Semblant a **C/C++,** GLSL suporta instruccions de control com **if**, **else**, **if/else**, **for**, **do-while**, **break**, **continue**, etc.

Les funcions definides per l'usuari també estan suportades, així com una amplia varietat de funcions predefinides, ja siguin comuns amb **C** i **C++** com **exp**(), **abs**(), … o bé relatives als gràfics com **ftransform**(), **smoothstep**() i **texture2D**().

Variables

La declaració i l'ús de variables també és similar a C i C++. Hi ha quatre opcions per qualificar una variable:

- **const** – Una constant.
- **varying** – Només de lectura al *fragment shader* i de lectura i escriptura al *vertex shader*. És utilitzada per comunicar paràmetres entre els dos *shaders*.
- **uniform** – Variable global només de lectura disponible tan per *vertex* com per *fragment shaders.*
- **attribute** – Variable global només de lectura només accessible pel *vertex shader*. Un atribut és passat des de el programa OpenGL al *shader*.

GLSL ens facilita moltes variables ja instanciades amb la informació de l'estat de OpenGL. Entre les accessibles pel *Vertex Shader* hi ha:

- **attribute variables**: *gl_Color* (color del vèrtex) , *gl_Normal* (valor de la normal del vèrtex), *gl_Vertex* (valor del vèrtex), ...
- **varying variables**: *gl_FrontColor*, *gl_BackColor*, ...
- **uniform variables**: *gl_ModelViewMatrix* (matriu ModelView), *gl_FrontMaterial* (material del vèrtex)
- **special output variables**: *gl_Position* (retorn de les coordenades del vèrtex, obligatòria)*, gl_PointSize*, ...

Entre els del *Fragment Shader* hi ha:

- **varying variables:** *gl_Color*, *glTexCoord[n]* (coordenada de la textura n), ...
- **special input variables:** *gl_FragCoord* (coordenada del fragment)
- **uniform variables**: *gl_ModelViewMatrix*, *gl_FrontMaterial*, ...
- **special output variables:** *gl_FragColor* (color de sortida del fragment), *gl_FragDepth*

Compilació i Execució

Els *shaders* GLSL no són aplicacions independents, sinó que requereixen una aplicació que usi l'API OpenGL. Els *shaders* GLSL són simplement un conjunt de cadenes de text que són passades al *driver* del fabricant de la targeta per a que sigui compilat, usant per això les crides adequades a l'API d'OpenGL.

Exemples

GLSL *Vertex Shader* Trivial: Assigna a cada Vertex la posició que li correspon després d'haver fet les transformacions de càmera i projecció.

```
void main(void)
{
gl_Position = ftransform();
}
```

GLSL *Fragment Shader* Trivial: Assigna a cada fragment generat el color vermell.

```
void main(void)
{
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

De manera que el que farà aquest *shader* és el que faria normalment la tarja gràfica però pintant-ho tot de vermell sense considerar ni materials, ni llums ni altres elements.

Cg

Cg o **C for Graphics** és una llenguatge de desenvolupament de *shaders* desenvolupat per NVIDIA en col·laboració amb Microsoft per programar *vertex* i fragment *shaders*. És molt similar per tan al llenguatge desenvolupat per Microsoft **HLSL**, que té la mateixa funcionalitat.

Cg, com GLSL, està basat en C i té la mateixa sintaxi, tot i que algunes característiques del C estan modificades i nous tipus de dades, també com a GLSL, han estat definits. El compilador de Cg proporciona *shaders* **DirectX** o **OpenGL**.

Bàsicament Cg té la mateixa funcionalitat que proveeix GLSL, encara que Cg requereix les seves biblioteques pròpies per tal de compilar i produir el codi dels *shaders* que s'usaran a la GPU de la tarja gràfica. El GLSL usa els *drivers* de la tarja gràfica per realitzar aquesta feina, i no requereix de biblioteques addicionals. Cal remarcar, que el **COLLADA FX**, que s'explica a l'apartat següent, usa **Cg** per generar els efectes dels materials o textures. És per això que el Cg és un requisit del present projecte.

Tipus de dades

Cg té els següents tipus de dades:

- **float** - nombre en coma flotant de 32 bits
- **half** – nombre en coma flotant de 16 bits
- **int** – enter de 32 bits
- **fixed** – nombre amb 12 bits de coma
- **bool** - booleà
- **sampler*** - representa una textura de * dimensions
- **float*** - vector * de floats
- **float*x*** - matriu de * per * de floats

Operadors, funcions i estructures de control

Com GLSL, suporta els operadors, funcions i estructures de control típics de C.

Exemple de Cg *vertex shader*

```
// input vertex
struct VertIn {
   float4 pos   : POSITION;
   float4 color : COLOR0;
 };

// output vertex
struct VertOut {
   float4 pos   : POSITION;
   float4 color : COLOR0;
 };

// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
   VertOut OUT;
   OUT.pos     = mul(modelViewProj, IN.pos); // calculate output coords
   OUT.color   = IN.color; // copy input color to output
   OUT.color.z = 1.0f; // blue component of color = 1.0f
   return OUT;
}
```

El que fa aquest *shader* és prendre la matriu d'estat d'OpenGL i multiplicar cada vèrtex per posicionar-lo correctament a l'escena. A més modifica el color de cada vèrtex per posar la component blava del color al màxim.

COLLADA

Pàgina Web: http://www.collada.org

A l'hora d'escollir un estàndard principal per carregar diferents models tridimensionals, es va pensar en primer moment en el format ***obj*** de **Maya**, i de fet les primeres versions de l'aplicació usaren aquest format per fer les primeres proves. Aviat però ens fixarem en **COLLADA**. COLLADA (**COLLA**borative **D**esign **A**ctivity) és un estàndard obert que estableix un esquema digital per a les aplicacions 3D interactives i és interessant perquè està recolzat per la indústria 3D, dissenyadors i desenvolupadors entre d'altres. Per això i les eines i biblioteques obertes que han sortit a la seva empara, el fa una elecció òptima per incloure un primer format de models tridimensionals.

COLLADA defineix un esquema XML de base de dades que possibilita que les diferents aplicacions de creació 3D puguin intercanviar informació fàcilment entre elles sense pèrdua d'informació. Això permet poder combinar diverses aplicacions per obtenir resultats òptims en modelats o dissenys complexos. Però COLLADA no és només un punt de trobada dels diferents estàndards de la indústria, sinó que ha aconseguit que els diferents fabricants incloguin suport directe a aquest estàndard. Així passa per exemple amb desenvolupadors d'aplicacions 3D, com **Autodesk** (**Maya** i **3D Studio**) i **Softimage** (**XSI**), que proporcionen importació i exportació directa a aquest estàndard.

L'esquema de COLLADA suporta totes les característiques modernes de les últimes aplicacions 3D, incloent efectes de *shaders* i simulació de física a través de **COLLADA FX** and **COLLADA Physics**. En concret la última versió de COLLADA, la 1.4, inclou entre altres característiques: geometria de models, *skinning*, *morphing*, animacions i validació de dades.

COLLADA FX és la primera definició de *shaders* en XML per a Collada que contempla tan la seva execució en sistemes que incloguin **GLSL** o bé **Cg**.

COLLADA Physics és la part encarregada de gestionar la física d'una escena i inclou l'intercanvi de dades amb hardware específic de gestió de física.

COLLADA DOM, publicada com a *open source*, és un *framework* per al desenvolupament d'aplicacions COLLADA i proporciona una interfície en C++ per carregar, consultar i guardar dades en l'estàndard COLLADA. El DOM possibilita treballar amb les dades de la base de dades XML (amb definició COLLADA) fent simples crides a funcions que representen aquesta base de dades. Les altres parts de COLLADA es basen en aquesta API.

L'aplicació desenvolupada tan sols aprofita una petita part de la potencialitat de COLLADA, tan sols el necessari per carregar un model triangulat senzill. L'objectiu del projecte, d'altra banda, no era desenvolupar un carregador sofisticat de COLLADA, ja que comportaria desenvolupar un projecte per si sol, donada la multitud de característiques que inclou. A més, aquesta part del projecte en un principi havia de trobar-se ja desenvolupada pel grup i només integrar-la a l'aplicació. Això al final no va ser possible, i es va optar per modificar un petit subconjunt de les biblioteques que venen amb COLLADA, en concret el projecte **COLLADA RT**. Aquest últim és el codi d'un renderitzador senzill d'escenes COLLADA en C++, que utilitza el COLLADA DOM i el COLLADA FX.

El projecte **COLLADA RT** s'ha modificat lleugerament per crear el que s'ha anomenat el projecte **ColladaParser**. Aquest projecte crea una biblioteca que serà usada per l'aplicació per extreure els models i la informació de llums i càmeres dels arxius **dae**, que són els que contenen informació COLLADA. Aquest projecte no és, com ja s'ha dit, n'hi molt menys un carregador exhaustiu i eficient, però ofereix una funcionalitat bàsica per proveir l'aplicació i acostar el projecte als seus objectius. Sens dubte, una versió més polida de l'aplicació hauria de millorar aquest aspecte.

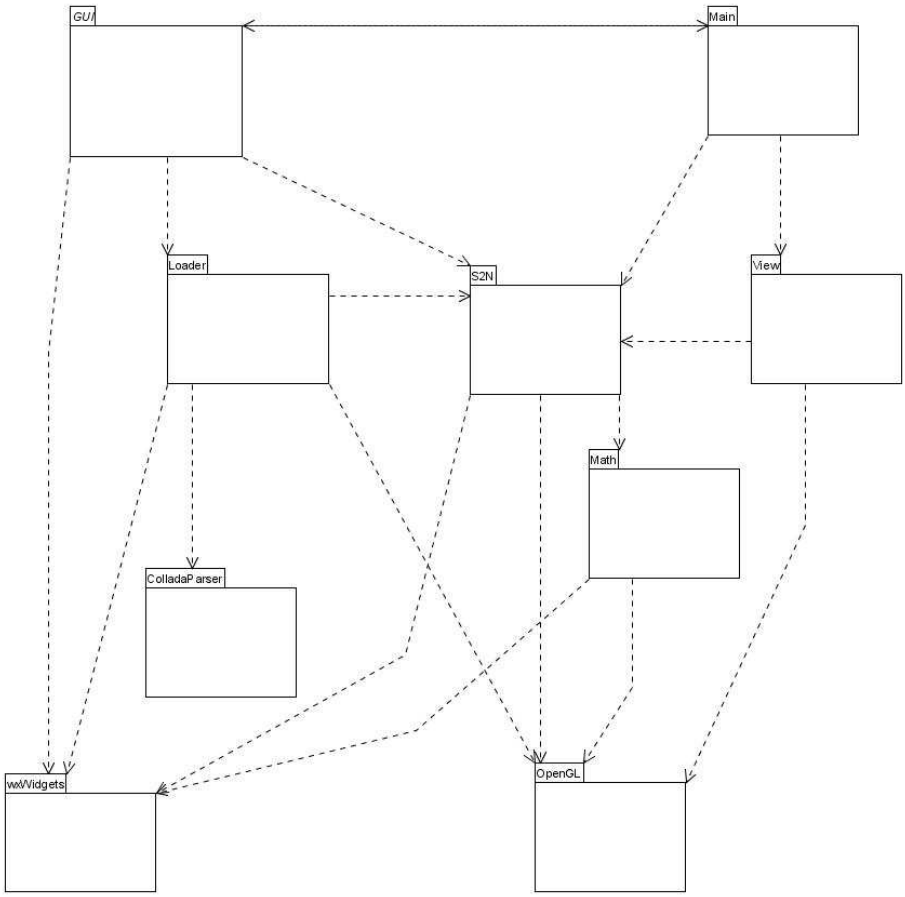
disseny de l'aplicació

L'explicació del disseny es basarà en mostrar com està estructurada l'aplicació en si, i obviarà les parts relatives al que s'ha explicat somerament anteriorment, com les **wxWidgets** i **COLLADA**. Pel volum mínim de canvis que conté **ColladaParser** (el projecte que defineix la llibreria per carregar els models COLLADA) respecte a COLLADA RT, d'on s'ha extret, remeto a la informació i documentació d'aquest projecte per clarificar-ho.

La mateixa consideració es fa per les classes, funcions i estructures de dades pertanyents a **wxWidgets**, que s'expliquen suficientment bé a la seva pròpia documentació.

diagrama de paquets

A continuació repassarem els paquets en que s'ha estructurat l'aplicació, que ajudaran a mostrar les relacions entre les classes creades. Aquest diagrama mostra les dependències de les classes que conté cada paquet.



explicació del diagrama

El paquet **GUI** és el que conté les classes encarregades de mostrar els elements per pantalla. És a dir, presentar els controls o *widgets* com panells, botons, finestres, barres, etc. Aquest paquet, a causa de la naturalesa del projecte (un entorn gràfic) està íntimament relacionat amb el paquet **Main. GUI** també usa classes de **Loader** per carregar els models tridimensionals i de **S2N** per presentar els valors dels elements de l'escena. Els elements que conformen la interfície gràfica, així com altres elements auxiliars, estan extrets de **wxWidgets**.

El paquet **Main**, com s'ha dit, està molt relacionat amb **GUI**, i té per missió inicialitzar l'aplicació i regir les configuracions ordenades des del GUI dels diferents Views, així com controlar els elements del GUI que s'han de dibuixar i refrescar. Aquest paquet usa classes de **S2N** per mantenir un seguiment de l'escena activa i classes de **View** per ordenar el repintat de les escenes usant els mòduls *View* indicats.

El paquet **Loader**, com ja s'ha apuntat, és l'encarregat de contenir les classes encarregades d'examinar els fitxers de models tridimensionals i convertir-los en les instàncies de **S2N** adients. Així és lògic que aquesta classe usi les classes de **S2N** per instanciar els elements carregats del fitxer. El paquet també utilitza el projecte **ColladaParser** per carregar els fitxers COLLADA. De fet, en la present versió aquests són els únics fitxers importables. Esmentar també que usa tangencialment el paquet **OpenGL** per usar-ne els tipus de dades.

El paquet **S2N** és aquell que recull les classes destinades a representar els diferents elements de l'escena i l'escena en si. Així, trobarem classes per representar les llums, càmeres, etc. i també la classe **S2N_Scene**, que emmagatzema tots els elements de l'escena i és referenciada per múltiples classes del projecte. Aquest paquet usa principalment els paquets: **OpenGL**, per utilitzar els seus tipus de dades, **Math**, per usar també els tipus dels Vectors definits allí, i **wxWidgets,** per usar-ne algunes funcions.

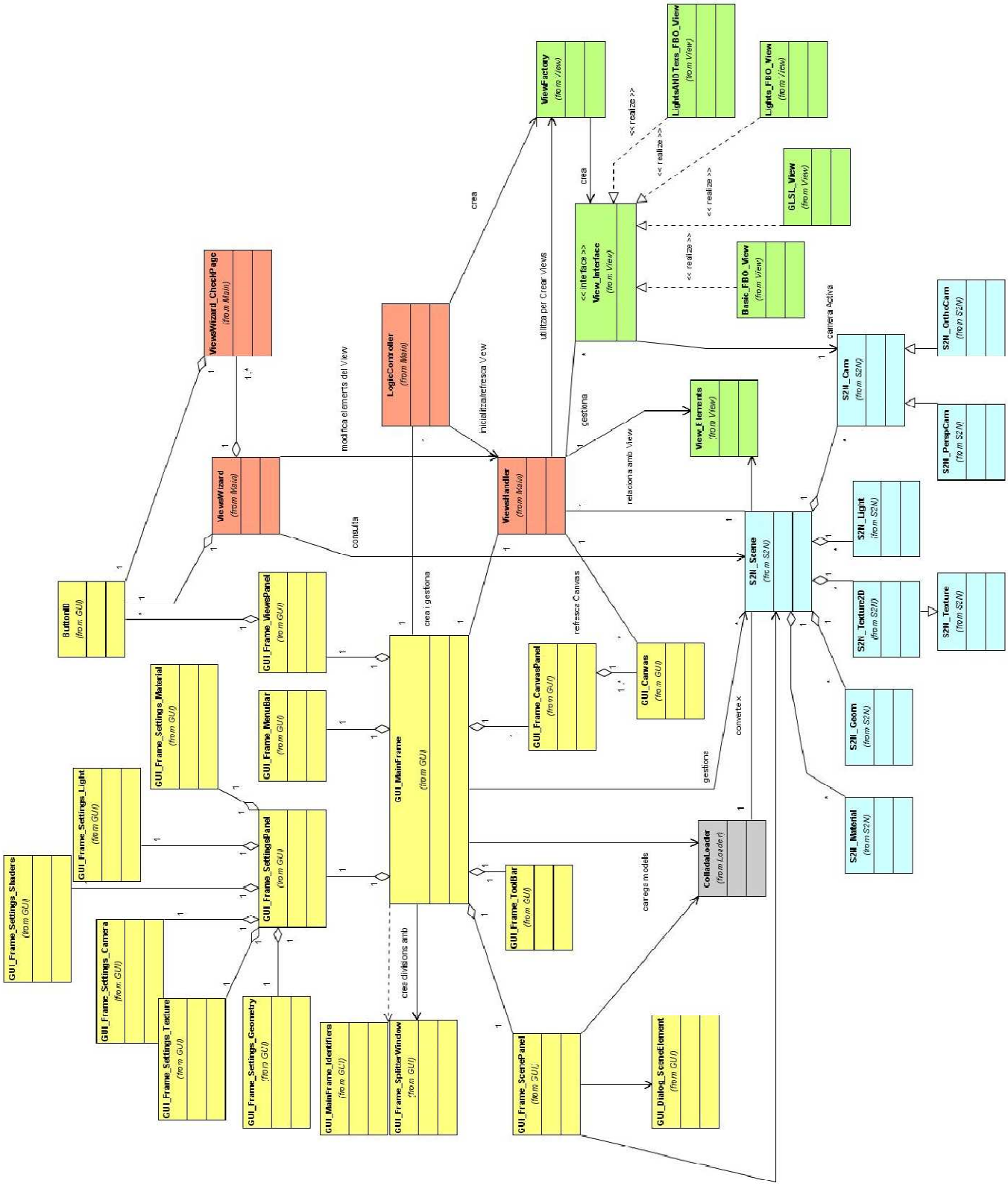
El paquet **Math** encapsula les classes dedicades a les classes representatives de tipus de dades matemàtiques, com són els Vectors. S'usen alguns tipus de dades de **OpenGL** i també algunes funcionalitats de **wxWidgets**.

ColladaParser no és, de fet, un paquet de l'aplicació, sinó que apareix al diagrama per representar el projecte del mateix nom, que implementa la llibreria responsable d'examinar un fitxer amb models COLLADA. És usat en exclusiva per la classe **ColladaLoader** del paquet **Loader**.

OpenGL és un paquet que conté les definicions de la llibreria GLEW, així com també alguns tipus de dades propis de l'aplicació relacionats amb l'estàndard OpenGL. Aquest pseudo-paquet, com l'anterior, també vol encapsular les crides a la llibreria OpenGL.

El paquet **wxWidgets**, com es pot deduir, tampoc és un paquet definit per l'aplicació, sinó que representa les crides de l'aplicació a funcions o a la derivació de classes al *framework* de wxWidgets.

diagrama de classes

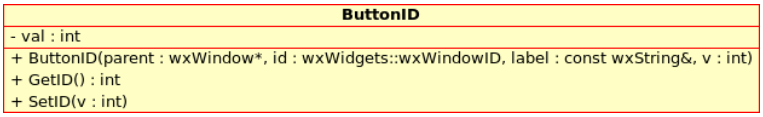


explicació del diagrama

Com es pot apreciar, en aquest diagrama no s'hi ha inclòs els paquets pertanyents a **OpenGL**, ni **Math**, ni **wxWidgets** ni **ColladaParser**. La raó és ben senzilla: tots aquests paquets són "accessoris", i no tenen rellevància real a l'hora d'explicar el disseny. Ans el contrari, farien més dificultosa la lectura del present diagrama, ja de per si amb forces elements. S'ha intentat així mateix no incloure símbols superflus ni reiteracions per no fer el diagrama més complex i intentar mantenir-ne el component explicatiu. Aquest diagrama, per la mateixa raó, no pretén representar totes i cadascuna de les associacions i dependències existents, sinó que s'ha d'interpretar com una guia per adonar-nos de les parts més rellevants del sistema.

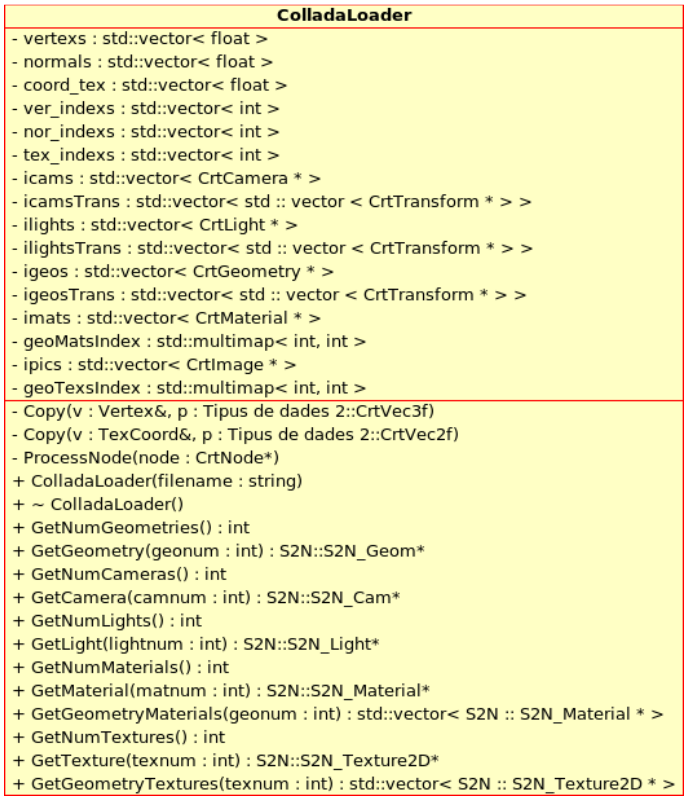
A continuació s'intentarà explicar els elements més interessant d'aquest disseny de classes, cas per cas.

ButtonID (GUI)



Crea un simple botó de wxWidgets on s'hi pot emmagatzemar un valor enter. Aquesta classe la utilitza **ViewsWizard** i **ViewsWizard_CheckPage** per crear la seva interfície. La relació d'una classe amb una altra és del tipus "formar part", com passa amb la majoria de les classes del paquet **GUI**, s'utilitzarà el símbol d'*agregació* 

ColladaLoader (Collada)



Classe que fa les crides necessàries per carregar un objecte Collada (*.dae). De moment només està pensat per objectes triangulats i sense animacions. Aquesta classe fa ús de la llibreria creada pel projecte **ColladaParser**, inclòs al projecte final.

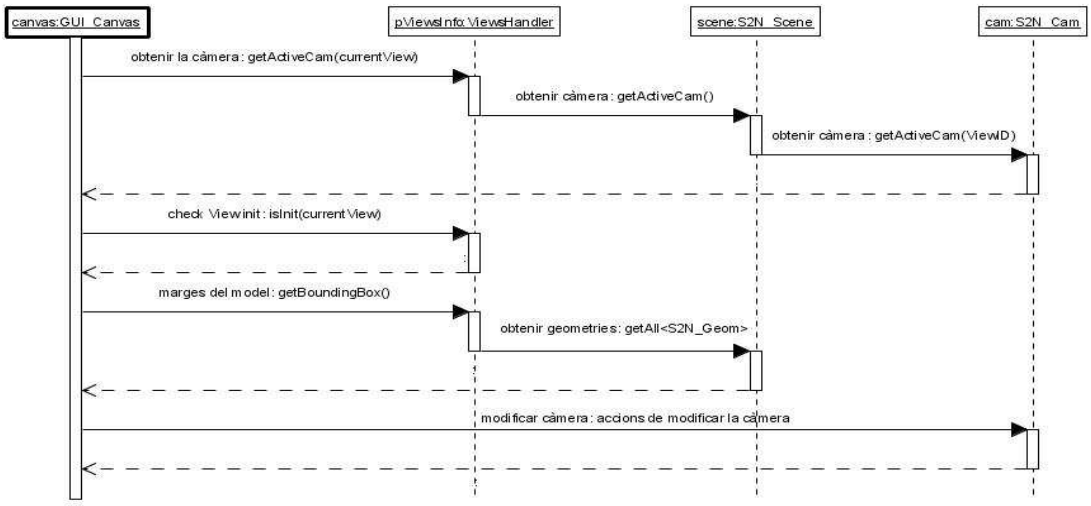
ColladaLoader és utilitzada, per carregar objectes COLLADA, per les classes **GUI_MainFrame** i **GUI_Frame_ScenePanel**. L'associació amb **S2N_Scene** vol representar el funcionament de la classe, que el que fa és retornar elements tals com **S2N_Geom**, **S2N_Light**, **S2N_Cam**, **S2N_Texture** i **S2N_Material**. Per no complicar el diagrama he optat per presentar una associació amb **S2N_Scene**, tot i que l'associació hauria de ser amb els elements abans esmentats.

GUI_Canvas
- init_gl : bool - type : Main::ViewsHandler::Canvas_Type - pViewsInfo : Main::ViewsHandler* - currentView : int - viewTexture : int - mouse : wxWidgets::wxPoint - mouseWheel : int - camtype : int
- initGL() - displayAxes() - reshape(width : int, height : int) + GUI_Canvas(parent : wxWindow*, id : wxWidgets::wxWindowID, typeIn : Main::ViewsHandler::Canvas_Type, vinfo : Main::ViewsHandler*, viewid : int, pos : const wxPoint&, size : const wxSize&, style : long, attribList : int*, name : const wxString&) + GUI_Canvas(parent : wxWindow*, id : wxWidgets::wxWindowID, typeIn : Main::ViewsHandler::Canvas_Type, vinfo : Main::ViewsHandler*, viewid : int, sharedContext : wxGLContext*, pos : const wxPoint&, size : const wxSize&, style : long, attribList : int*, name : const wxString&) + display() + getTexture() : int + getView() : int + getCam() : S2N::S2N_Cam* + getType() : int + setParams(tex : int, canvas_type : int, viewID : int) + clean() + OnPaint(event : wxPaintEvent&) + OnIdle(event : wxIdleEvent&) + OnSize(event : wxSizeEvent&) + OnMouse(event : wxMouseEvent&) + OnKeyDown(event : wxKeyEvent&) + OnErase(event : wxEraseEvent&)

GUI_Canvas (GUI)

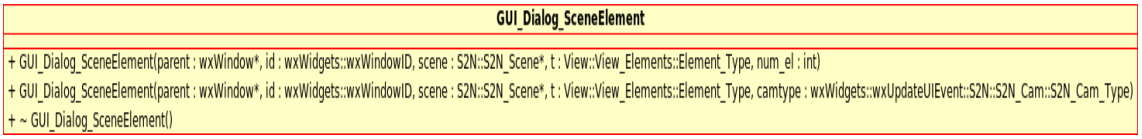
La seva funció principal és rebre un identificador de textura d'OpenGL i mostrar-nos aquesta textura. Les accions o moviments indicats sobre el Canvas són passats al *View* corresponent per tal d'actualitzar la textura que estem veient.

Tal com es veu al diagrama, la classe és usada per **GUI_Frame_CanvasPanel**, que té cinc *canvas*: un de gran i quatre de petits per fer les vistes de costat. La classe **ViewsHandler** és la que mana als *canvas* que es refresquin usant els elements definits pel *View*. Per il·lustrar aquest comportament, detinguem-nos un moment a veure el funcionament de la funció **OnMouse()**.



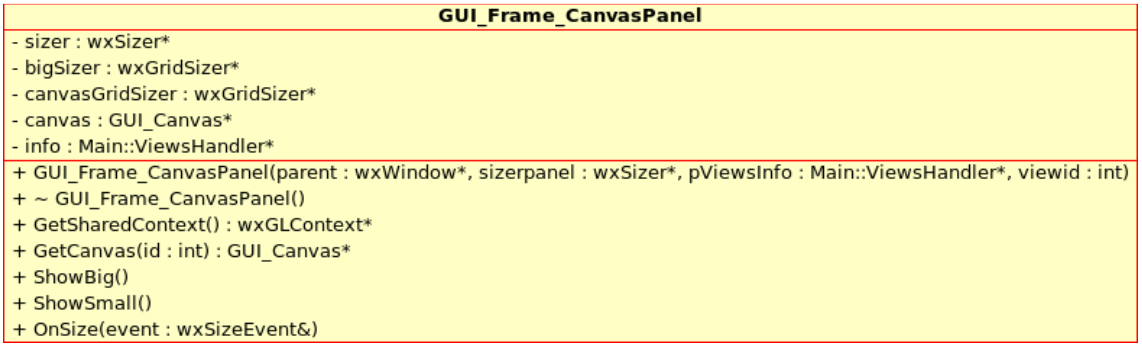
Aquesta funció, que rep les interrupcions del ratolí, el primer que fa és obtenir la càmera activa del *View*. L'encarregat d'això és l'objecte de la classe **ViewsHandler**, que manté una relació dels elements de l'escena definits per cada *View.* **ViewsHandler** pregunta al seu torn a la classe **S2N_Scene**, que conté totes les càmeres. La funció **OnMouse()**, comprova després si el *View* està inicialitzat, i seguidament obté els marges màxims del model a l'espai, per tal d'escalar els moviments fets amb el *mouse* a l'espai del model. Finalment modifiquem la càmera amb els moviments escalats de la càmera. No cal dir que la funció que gestiona els moviments fets amb el teclat té exactament el mateix comportament.

GUI_Dialog_SceneElement (GUI)



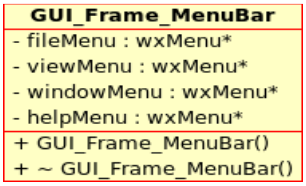
El diàleg deriva de la classe **wxDialog** de **wxWidgets**. Segons el tipus que passem al constructor, defineix uns elements o uns altres a la finestra. Aquesta classe és cridada per la classe **GUI_Frame_ScenePanel**, que és el panell on es mostren els elements de l'escena, quan cliquem sobre un element de l'escena.

GUI_Frame_CanvasPanel (GUI)



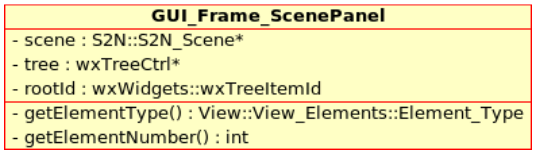
Aquest panell és un element del GUI que conté en total cinc canvas, un de gran (BigCanvas) i quatre de petits per les vistes de costat (Classe **GUI_Canvas)**. És l'encarregat de iniciar-los, fent-ho de manera que comparteixen entre si un sol context de **OpenGL**.

GUI_Frame_MenuBar (GUI)



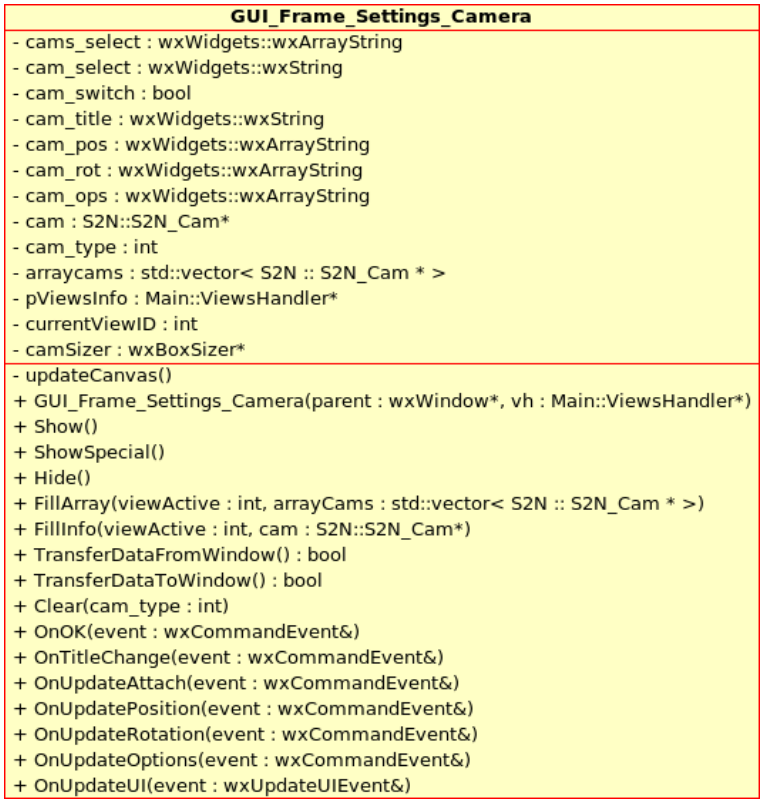
Component de la finestra principal de l'aplicació que defineix els elements de la barra de menú de l'aplicació. Té una relació 1 a 1 amb la finestra principal.

GUI_Frame_ScenePanel (GUI)

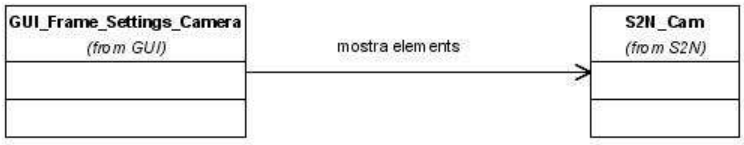


Classe derivada d'un panell de **wxWidgets** que forma part de la finestra principal de l'aplicació (**GUI_MainFrame**). Conté l'arbre on es situen els elements d'escena carregats a l'aplicació, tals com geometries, llums, ... Clicant sobre els elements, en podrem veure i modificar les propietats, a través de la classe **GUI_Dialog_SceneElement**.

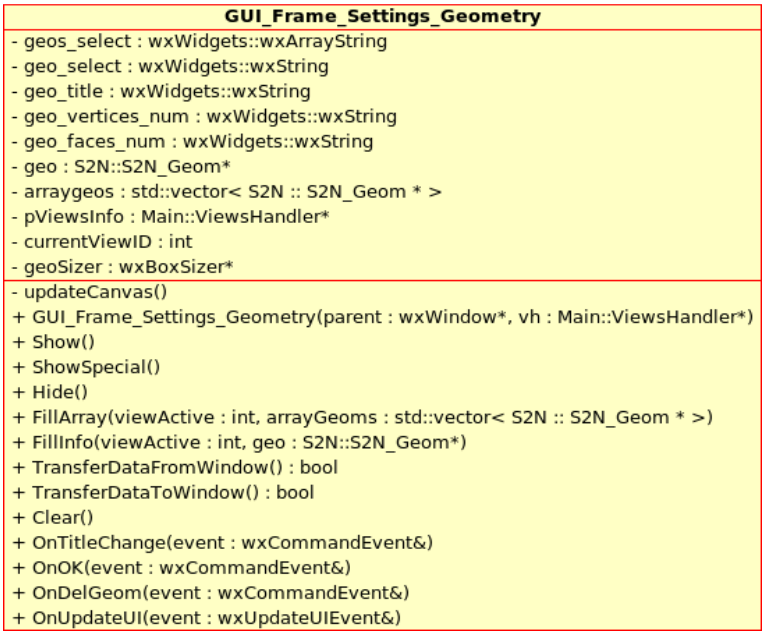
GUI_Frame_Settings_Camera (GUI)



Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar i manipular les propietats d'una càmera. Disposa de mètodes per transferir la informació de la càmera als controls i viceversa. Forma part del panell **GUI_Frame_SettingsPanel**. Tan aquest panell com els altres que segueixen a continuació estan relacionats també, evidentment, amb la classe que representen (en aquest cas **S2N_Cam**). Aquesta relació no s'ha explicitat al diagrama per no fer-lo il·legible.

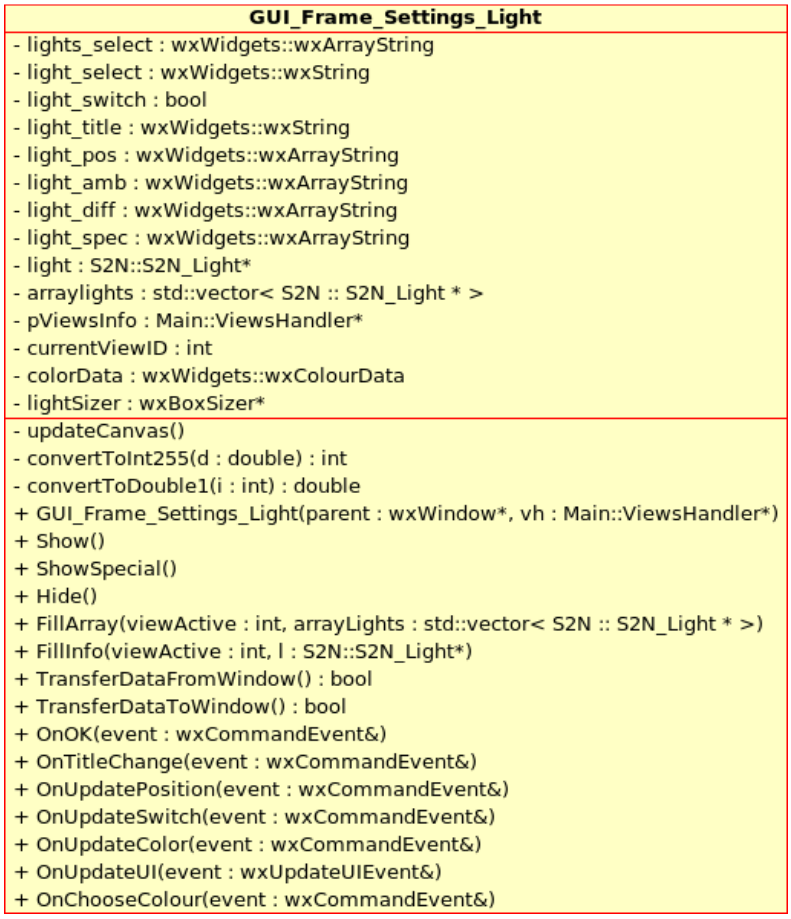


GUI_Frame_Settings_Geometry (GUI)



Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar les propietats de la geometria. Disposa de mètodes per transferir la informació de la geometria als controls. Forma part del panell **GUI_Frame_SettingsPanel**.

GUI_Frame_Settings_Light (GUI)

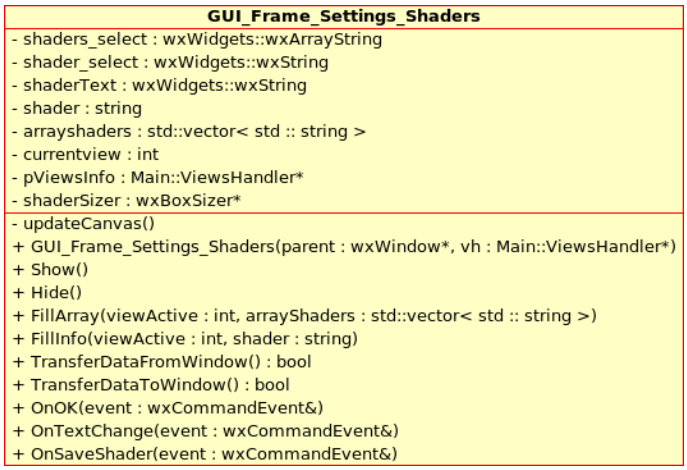


Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar i manipular les propietats d'una llum. Disposa de mètodes per transferir la informació de la llum als controls i viceversa. Forma part del panell **GUI_Frame_SettingsPanel**.

GUI_Frame_Settings_Material (GUI)

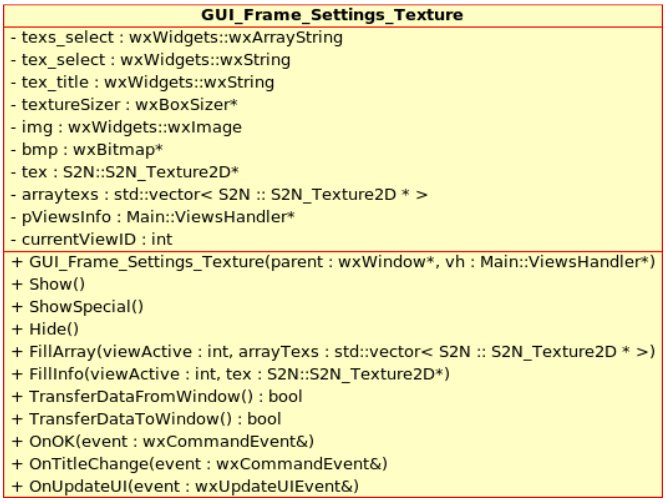
Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar i manipular les propietats d'un material. Disposa de mètodes per transferir la informació del material als controls i viceversa. Forma part del panell **GUI_Frame_SettingsPanel**.

GUI_Frame_Settings_Shaders (GUI)

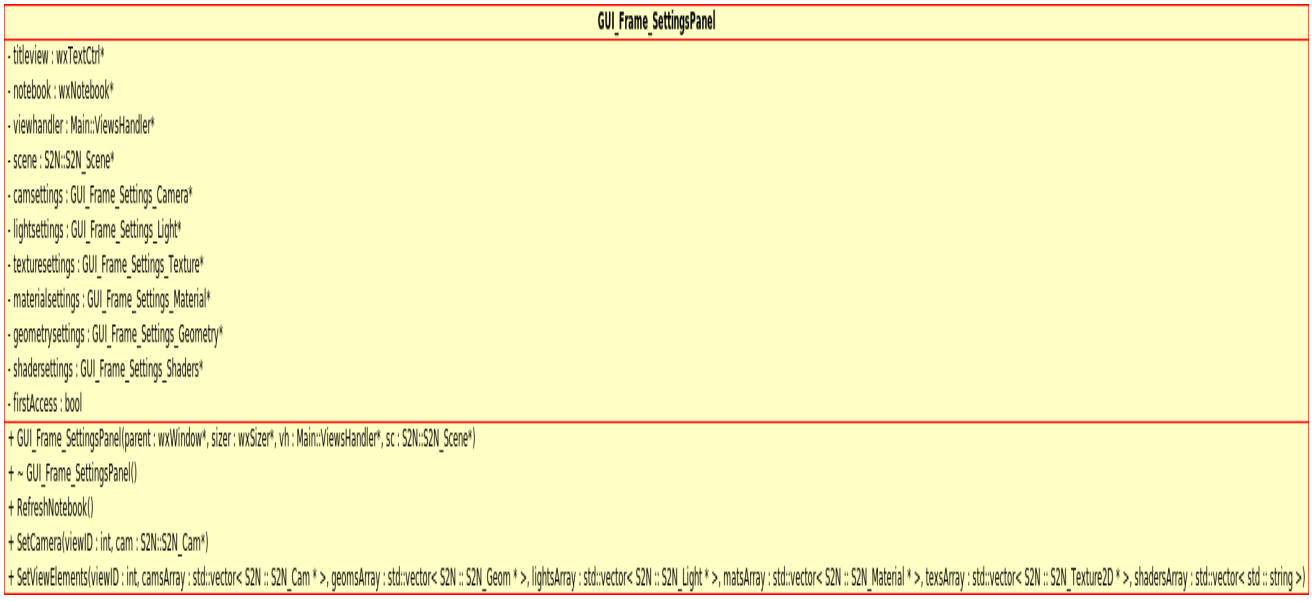


Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar el contingut d'un shader. Disposa de mètodes per transferir la informació del *shader* al control i viceversa. Forma part del panell **GUI_Frame_SettingsPanel**.

GUI_Frame_Settings_Texture (GUI)



Deriva d'un panell de **wxWidgets** i disposa els elements d'interfície per mostrar i manipular les propietats d'una llum. Disposa de mètodes per transferir la informació de la llum als controls i viceversa. Forma part del panell **GUI_Frame_SettingsPanel**.



GUI_Frame_SettingsPanel (GUI)

Defineix un llibre (amb pestanyes per cada categoria d'element), on podrem navegar pels diferents elements visualitzant-los i canviant-los. Està format per tots els elements definits anteriorment (**GUI_Frame_Settings_***) i forma part al seu torn de la finestra principal de l'aplicació (**GUI_MainFrame**).

GUI_Frame_SplitterWindow (GUI)

GUI_Frame_SplitterWindow
- sash : int
+ GUI_Frame_SplitterWindow(p : wxWindow*, pixelsSash : int, gravity : float, miniumpanelsize : int) + Split(win1 : wxWindow*, win2 : wxWindow*) + ReplacePanel(win : wxWindow*) + OnPositionChanged(event : wxSplitterEvent&) + OnPositionChanging(event : wxSplitterEvent&) + OnUnsplitEvent(event : wxSplitterEvent&)

Ens permet dividir una finestra en dues parts, dividides per una línia intermèdia mòbil, que permet redimensionar les dues finestres creades. Aquesta classe la usa la classe de la finestra principal de l'aplicació (**GUI_MainFrame**).

GUI_Frame_ToolBar (GUI)

GUI_Frame_ToolBar
+ GUI_Frame_ToolBar(parent : wxWindow*, id : wxWidgets::wxWindowID, pos : const wxPoint&, size : const wxSize&) + ~ GUI_Frame_ToolBar()

Defineix la barra d'eines de la finestra amb una sèrie de botons ja predefinits. Deriva de la classe de **wxWidgets wxToolBar** i és usada per la finestra principal (classe **GUI_MainFrame**).

GUI_Frame_ViewsPanel (GUI)

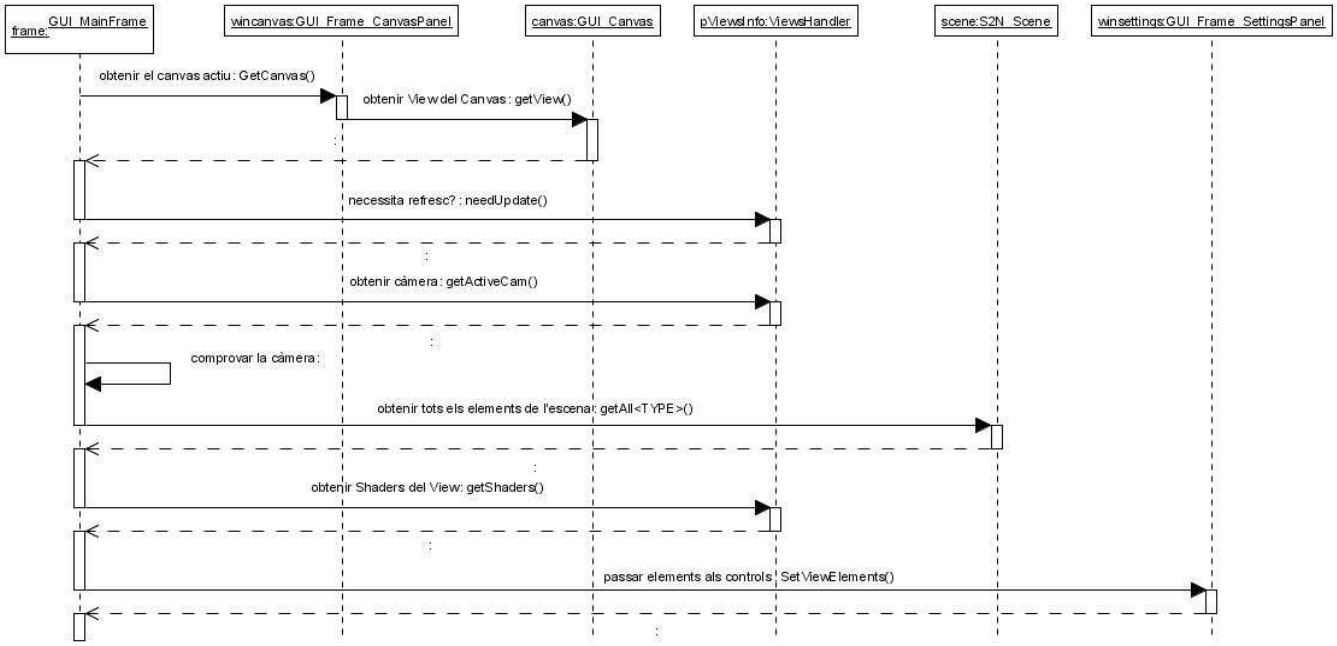
GUI_Frame_ViewsPanel
- togglebutton : wxToggleButton* - scene : S2N::S2N_Scene* - viewhandler : Main::ViewsHandler* - ViewID_togglebtnID : std::map< int, int >
+ GUI_Frame_ViewsPanel(parent : wxWindow*, sizer : wxSizer*, sc : S2N::S2N_Scene*, vh : Main::ViewsHandler*) + ~ GUI_Frame_ViewsPanel() + RefreshButtons(views_inits : std::vector< int >) + setButton(ID : int, value : bool) + setActive(ID : int) + OnButtonConfigure(event : wxCommandEvent&) + OnButtonConfigureOne(event : wxCommandEvent&)

Amb aquest panell, que conforma també la finestra principal de l'aplicació (classe **GUI_MainFrame**), podrem configurar els diferents Views disponibles. Un cop configurats amb els elements que necessiten cadascun, els podrem posar en marxa pitjant el botó corresponent. Està format per botons de la classe **ButtonID.**

GUI_MainFrame (GUI)

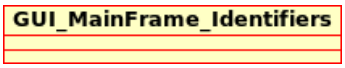
GUI_MainFrame
- init : bool - rootsizer : wxBoxSizer* - canvasSizer : wxBoxSizer* - vistesSizer : wxBoxSizer* - settingsSizer : wxBoxSizer* - sceneSizer : wxBoxSizer* - splitter : GUI_Frame_SplitterWindow* - barraMenu : GUI_Frame_MenuBar* - barraTools : GUI_Frame_ToolBar* - winsettings : GUI_Frame_SettingsPanel* - wincanvas : GUI_Frame_CanvasPanel* - winviews : GUI_Frame_ViewsPanel* - winscene : GUI_Frame_ScenePanel* - activeCanvas : GUI::GUI_MainFrame_Identifiers::canvasType - pViewsInfo : Main::ViewsHandler* - scene : S2N::S2N_Scene*
- Set1Canvas() - Set4Canvas() + GUI_MainFrame(title : const wxChar*, xpos : int, ypos : int, width : int, height : int, vi : Main::ViewsHandler*, scene : S2N::S2N_Scene*) + ~ GUI_MainFrame() + GetScene() : S2N::S2N_Scene* + GetContext() : wxGLContext* + GetNumberOfCanvas() : int + GetCanvasView(t : GUI::GUI_MainFrame_Identifiers::canvasType) : int + GetCanvasCameraType(t : GUI::GUI_MainFrame_Identifiers::canvasType) : int + GetCanvasTexture(t : GUI::GUI_MainFrame_Identifiers::canvasType) : int + SetCanvasParams(t : GUI::GUI_MainFrame_Identifiers::canvasType, id_view : int, id_tex : int) + SetActiveCanvas(ac : GUI::GUI_MainFrame_Identifiers::canvasType) + GetActiveCanvas() : GUI_Canvas* + IsMultiView() : bool + RefreshCanvas(ac : GUI::GUI_MainFrame_Identifiers::canvasType) + OnOneView(event : wxCommandEvent&) + OnFourView(event : wxCommandEvent&) + OnToolBarEnter(event : wxCommandEvent&) + OnViewScene(event : wxCommandEvent&) + OnViewViews(event : wxCommandEvent&) + OnViewSettings(event : wxCommandEvent&) + OnViewNothing(event : wxCommandEvent&) + OnToggleView(event : wxCommandEvent&) + OnSize(event : wxSizeEvent&) + OnIdle(event : wxIdleEvent&) + OnBackgroundErase(event : wxEraseEvent&) + OnOpenScene(event : wxCommandEvent&) + OnCloseScene(event : wxCommandEvent&) + OnOpenMesh(event : wxCommandEvent&) + OnSaveScene(event : wxCommandEvent&) + OnRunWizard(event : wxCommandEvent&) + OnAbout(event : wxCommandEvent&) + OnExit(event : wxCommandEvent&)

Classe que defineix la finestra principal de l'aplicació, que deriva de la classe **wxFrame** de **wxWidgets**. Aquesta és la finestra encarregada de cridar les classes pertinents (les explicades fins ara), que crearan la interfície principal de l'aplicació, amb els menús, els panells, etc. Aquesta classe usa també **ColladaLoader** per carregar els models COLLADA a l'escena. Així mateix la classe utilitza també **S2N_Scene**, per carregar els fitxers amb informació d'elements d'escena i per passar la informació de l'escena a les classes del GUI que ho necessiten. També manté una relació amb la classe **ViewsHandler** per assabentar-se dels elements de la interfície que cal refrescar. Veiem un diagrama d'aquesta última funcionalitat, representada per la funció **OnIdle()** :



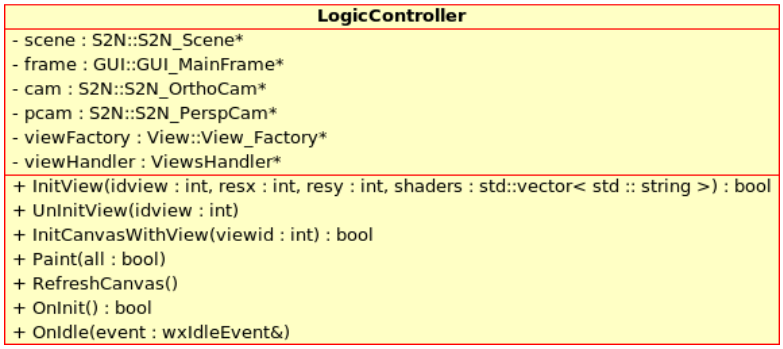
El primer que fem és obtenir el *View* actiu a través de la classe **GUI_Frame_CanvasPanel**, preguntant aquesta última classe a **GUI_Canvas**, que retornarà el valor. Preguntem a la classe **ViewsHandler** si el *View* necessita repintar-se i n' obtenim d'aquesta també la càmera activa per comprovar si la càmera està definida i cal o no continuar l'execució. En el cas que la càmera estigui definida, obtenim tots els elements de l'escena associats al *View*, així com els *shaders*, per passar-los finalment a la classe **GUI_Frame_SettingsPanel**, que és la encarregada de mostrar la informació dels elements de l'escena als controls gràfics.

GUI_MainFrame_Identifiers (GUI)



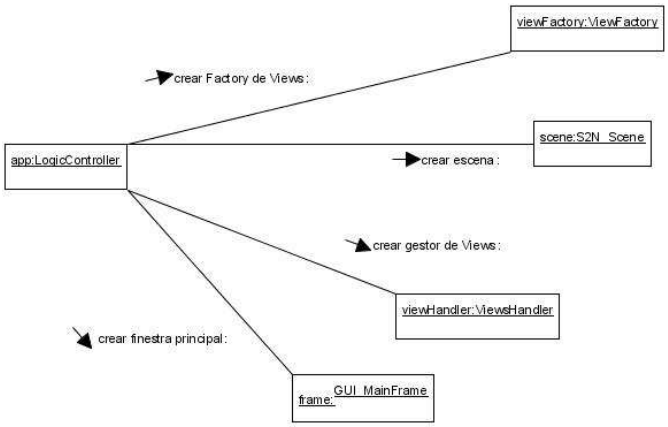
Defineix els identificadors dels events que usen diverses finestres del GUI, en especial la finestra principal (classe **GUI_MainFrame**) i les classes del GUI associades a aquesta. També conté les referències a les icones usades.

LogicController (Main)

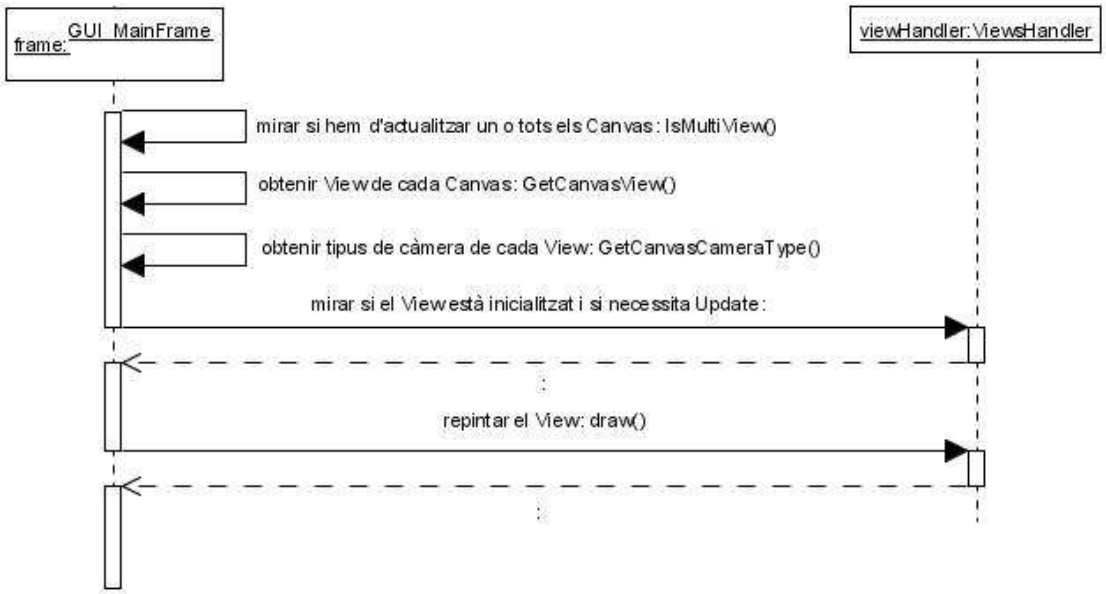


Aquesta classe és la classe inicial de l'aplicació i deriva de la classe **wxApp** de **wxWidgets**. Té per missió crear la finestra principal (i així tot el GUI), i gestionar els diferents *Views* programats. Aquesta gestió consisteix a iniciar-los i obligar-los a repintar-se periòdicament, per tal de mantenir-los actualitzats. Així doncs podem veure les relacions que té amb **GUI_MainFrame**, amb **ViewsHandler** i amb **ViewFactory**. Per ser la classe que inicia l'aplicació,

mereix un esment especial. Veiem uns quant diagrames que il·lustren diversos aspectes de la classe.

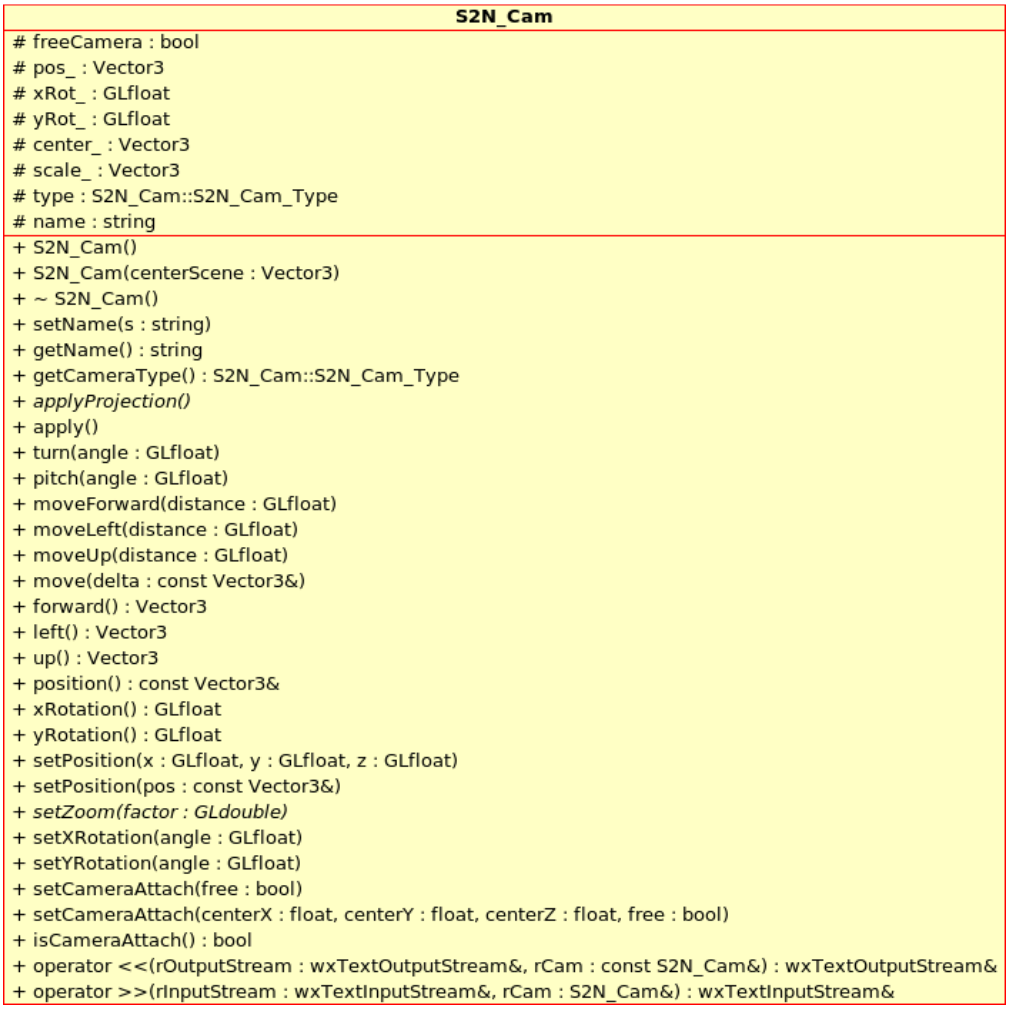


Aquest primer diagrama representa les classes que instancia **LogicController** al iniciar-se, a través del seu mètode **OnInit()**. Aquestes instàncies són úniques, i es mantenen durant tota l'execució de l'aplicació.



Aquest diagrama mostra la col·laboració entre la classe **GUI_MainFrame** i **ViewsHandler**, concretament al mètode **Paint()** del primer. Aquest mètode és llançat per l' *idle* del sistema, i s'encarrega, tal com es veu, de cridar **ViewsHandler** i el seu mètode **draw()** per cada *canvas*.

S2N_Cam (S2N)

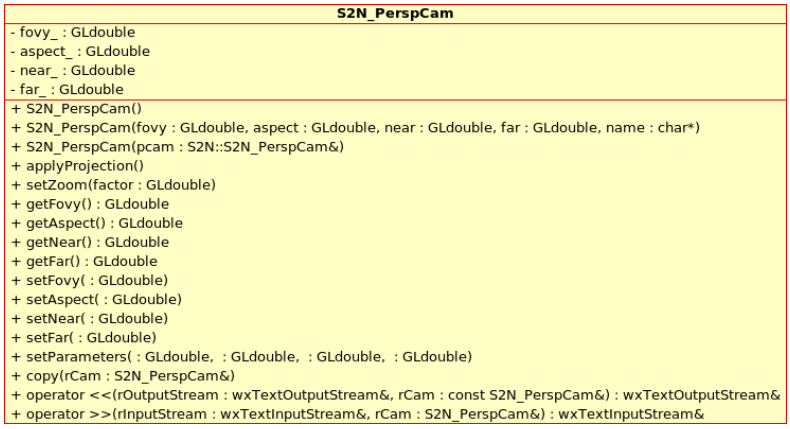


Defineix una classe que defineix els elements d'una càmera, tals com posició, rotacions, ... També defineix mètodes per aplicar les ordres OpenGL relatives a la càmera. La classe **S2N_Cam** és usada per moltes altres, però vull destacar la relació que té amb **Views_Interface**, que la utilitza per establir la càmera que s'usarà per pintar un cert *View*. La classe també forma part de **S2N_Scene** i és usada per altres com **GUI_Canvas i GUI_Frame_Settings_Camera**, entre d'altres, ja que aquesta és una de les classes més usades al projecte i hi ha moltes referències a ella.

S2N_OrthoCam (S2N)

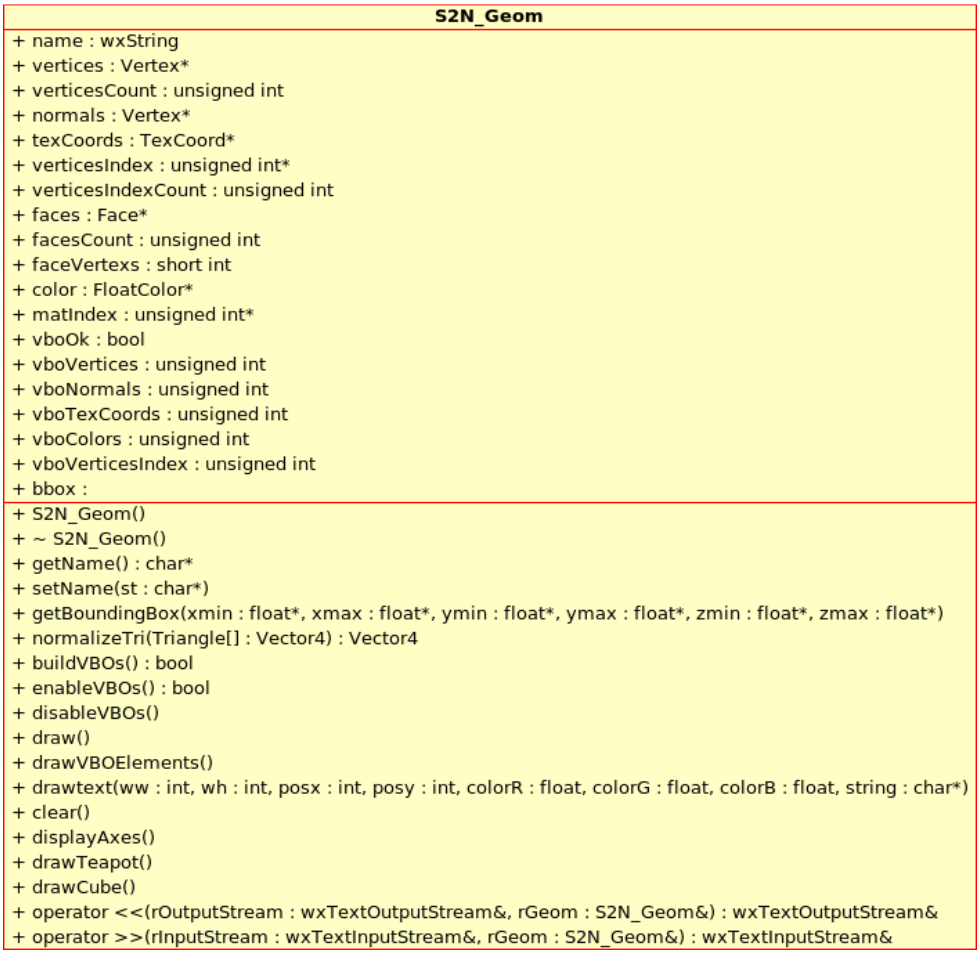
Defineix una classe, derivada de **S2N_Cam**, que defineix els elements d'una càmera ortogonal, tals com posició, rotacions, ... així com els límits del prisma que defineix la càmera ortogonal. També defineix mètodes per aplicar les ordres OpenGL relatives a la càmera.

S2N_PerspCam (S2N)



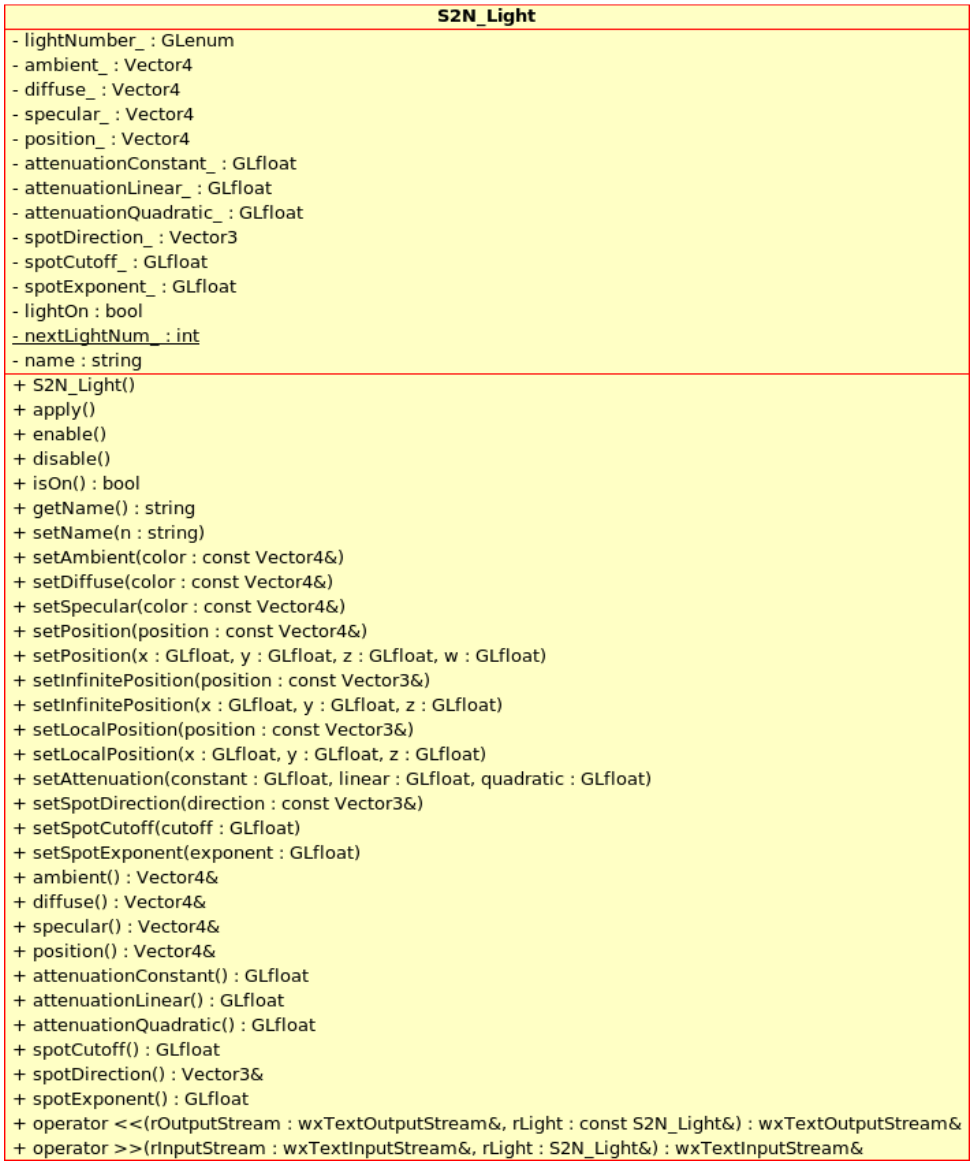
Defineix una classe, derivada de **S2N_Cam**, que defineix els elements d'una càmera en perspectiva, tals com posició, rotacions, angle d'obertura, aspecte, ... així com els límits del de la càmera. També defineix mètodes per aplicar les ordres OpenGL relatives a la càmera.

S2N_Geom (S2N)



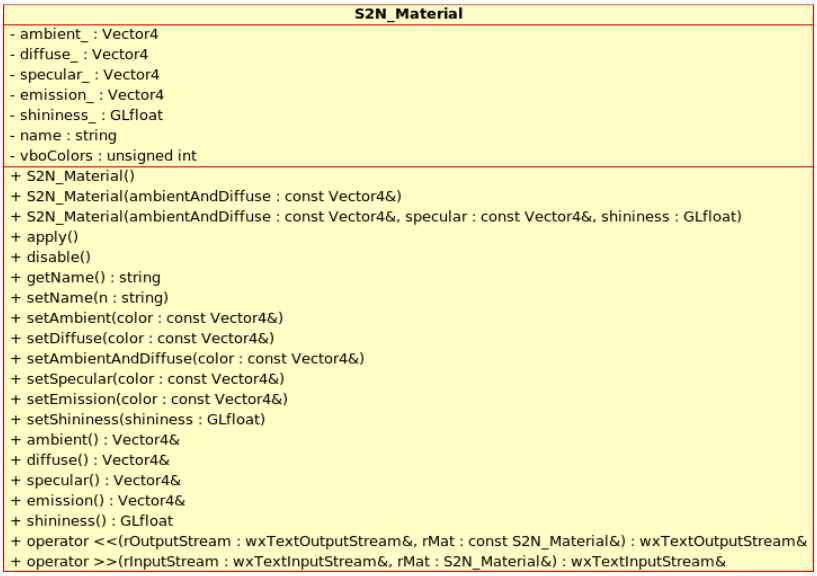
Emmagatzema les dades d'un objecte tridimensional i ofereix alguns mètodes per tal d'automatitzar feines repetitives. Per raons de compatibilitat, es mantenen les variables com a públiques, permetent l'accés directe als atributs de la classe. Forma part de la classe **S2N_Scene**.

S2N_Light (S2N)



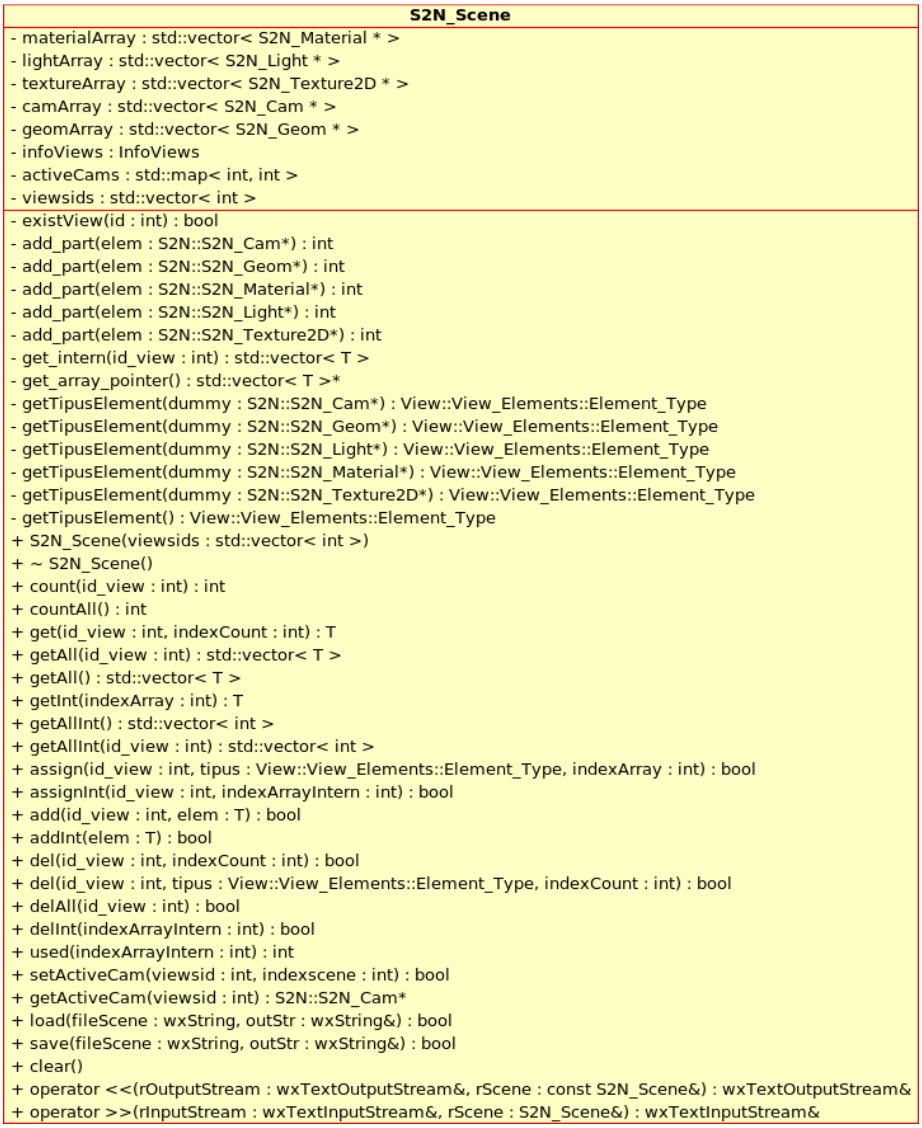
Aporta l'emmagatzematge i l'aplicació de mètodes per manipular una llum d'OpenGL. Forma part de la classe **S2N_Scene**.

S2N_Material (S2N)



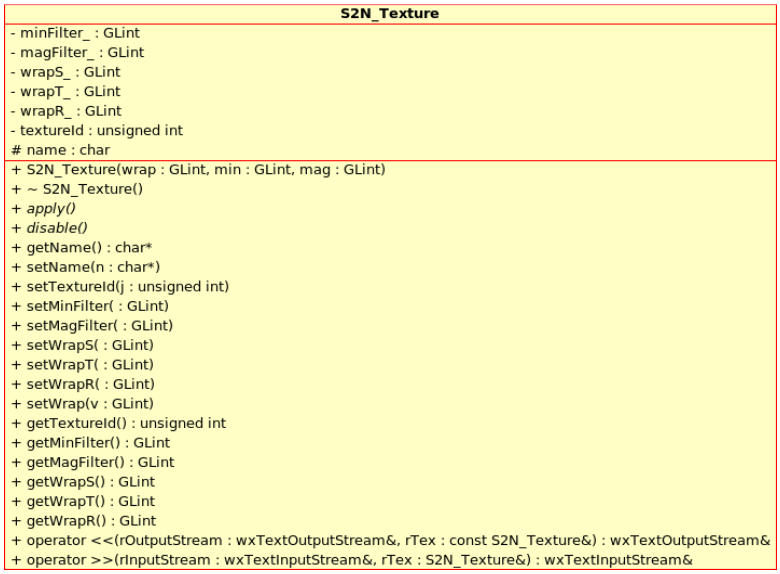
Guarda les propietats d'un material i proveeix mètodes per accedir a les característiques d'aquest material OpenGL. Forma part de la classe **S2N_Scene**.

S2N_Scene (S2N)



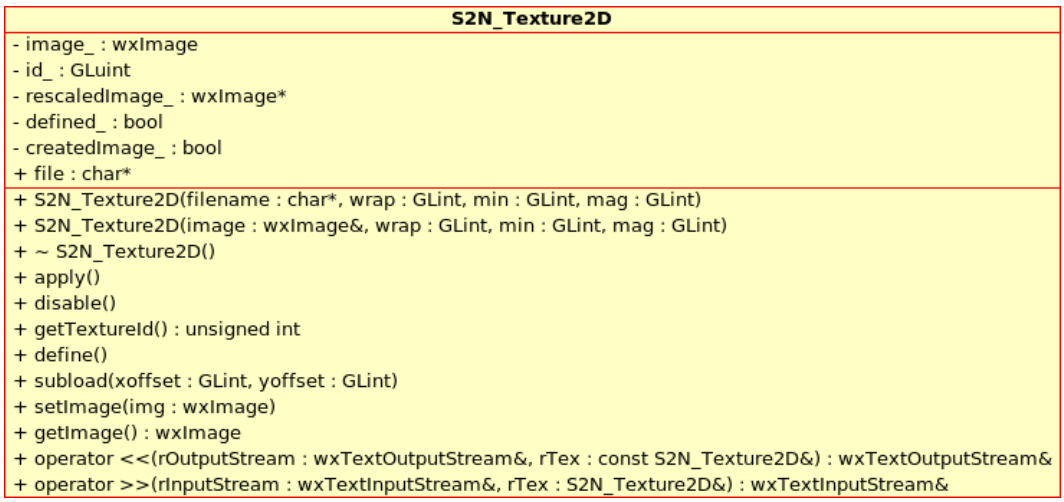
Una de les classes més usades del projecte, controla els elements carregats de l'escena. Aquests elements poden estar o no assignats a un determinat *View*. La classe proveeix les estructures i mètodes necessaris per assignar, cercar, desassignar, ... els elements de l'escena dels diferents *Views*. **S2N_Scene** està relacionat amb **ColladaLoader**, que usa de fet els seus components (**S2N_Cam, S2N_Light, S2N_Material, S2N_Texture2D, s2n_Geom**) per retornar objectes de les classes utilitzables pel projecte a partir d'objectes COLLADA. La classe **ViewsWizard** usa aquesta classe per presentar els elements de l'escena, i així mateix ho fa **GUI_Frame_ScenePanel**. La classe **GUI_MainFrame** carrega a la classe els elements extrets dels fitxers d'escena. **ViewsHandler** relaciona **S2N_Scene** i **View_Interface**, mantenint els elements de l'escena per cada realització de *View*.

S2N_Texture (S2N)



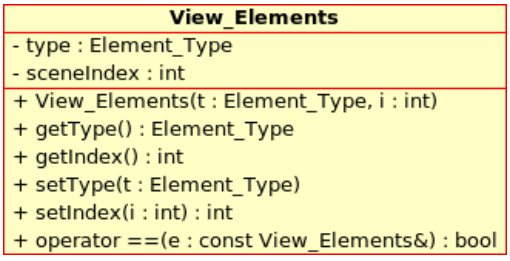
Classe Abstracta que guarda les propietats d'una textura i proveeix mètodes per accedir a les característiques d'aquesta. És la classe base de **S2N_Texture2D**.

S2N_Texture2D (S2N)



Classe derivada de **S2N_Texture** que guarda les propietats d'una textura 2D i proveeix mètodes per accedir a les característiques d'aquesta. Forma part de la classe **S2N_Scene**.

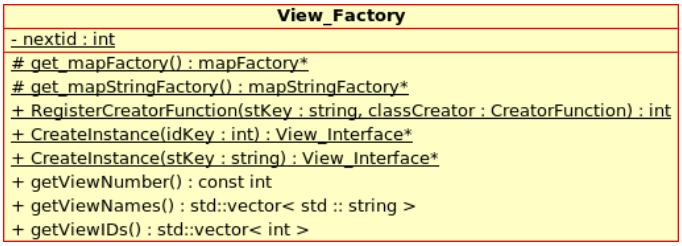
View_Elements (View)



Defineix els tipus dels elements que podem tenir en un *View*, guardant la parella del tipus d'element i la posició al vector pertinent de l'escena. És usada per **S2N_Scene** per mantenir els

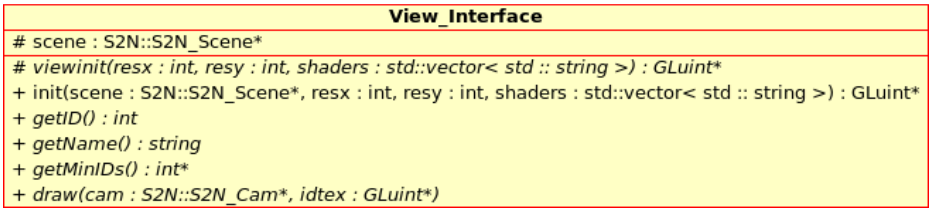
tipus d'elements de l'escena i per **ViewsHandler** per saber a quins *Views* corresponen els elements de l'escena.

View_Factory (View)



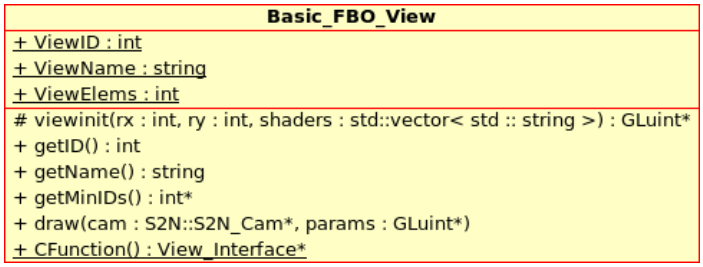
Gestiona la creació de nous *Views*, actuant com a *Factory*. Aquest tipus de classes ajuden a la creació i al maneig de classes derivades, com els *Views*, que deriven de **View_Interface**. Així doncs aquesta classe és instanciada i usada per **LogicController** per instanciar els diferents *Views* i també per **ViewsHandler** per extreure'n informació del número de *Views*, dels noms, ...

View_Interface (View)



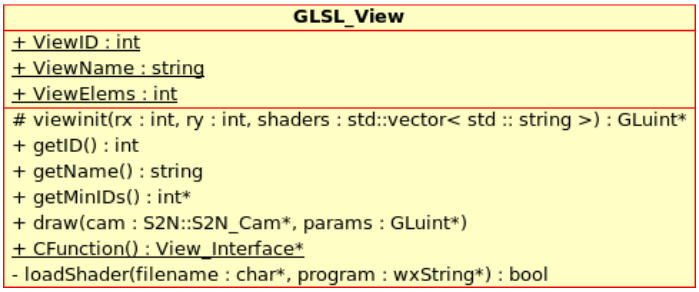
Interface d' on hereten els *Views*. Aquesta classe estableix els mètodes i variables que hauran de tenir tots els *Views*, de la qual deriven. Aquesta classe també està relacionada amb **S2N_Cam**, ja que defineix una càmera activa per cada *View*. La classe **ViewsHandler** gestiona les classes derivades de **View_Interface**.

Basic_FBO_View (View)



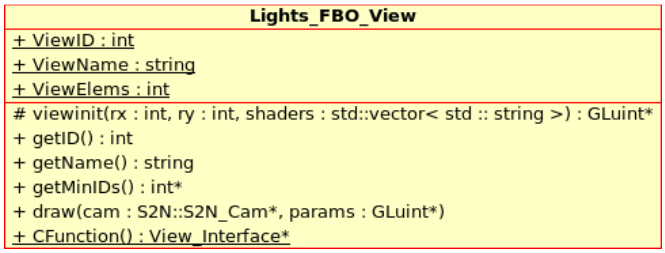
Producte concret de la *Factory* de *Views* que utilitza **Frame Buffer Objects** per renderitzar només la geometria, sense tenir en compte llums, materials, ...

GLSL_View (View)



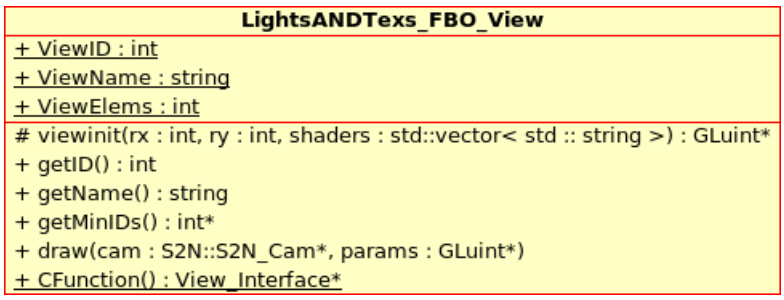
Producte concret de la *Factory* de *Views* que utilitza **GL Shading Language** per millorar la renderització.

Lights_FBO_View (View)



Producte concret de la *Factory* de *Views* que utilitza **FBOs** per renderitzar la geometria, tenint només en compte els llums.

LightsANDTexs_FBO_View (View)



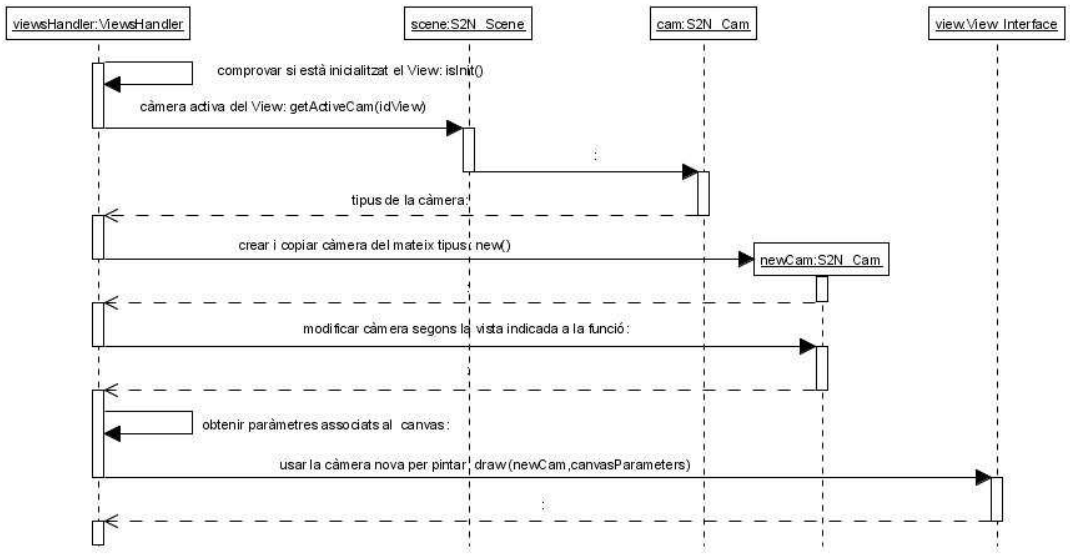
Producte concret de la *Factory* de *Views* que utilitza **FBOs** per renderitzar la geometria, tenint en compte llums, materials i textures.

ViewsHandler (Main)

ViewsHandler
- info : std::map< int, oneviewinfo > - actual : std::map< int, oneviewinfo >::iterator - viewFactory : View::View_Factory* - scene : S2N::S2N_Scene*
+ ViewsHandler(numcanvas : int, viewFactory : View::View_Factory*, scene : S2N::S2N_Scene*) + ~ ViewsHandler() + getScene() : S2N::S2N_Scene* + initViewCanvas(viewsid : int, canvasid : int, canvastype : int, resX : int, resY : int, shaders : std::vector< std :: string >) : bool + getAllIDs() : std::vector< int > + getAllNames() : std::vector< std :: string > + setInit(viewid : int, set : bool) : bool + isInit(viewid : int) : bool + needUpdate(viewid : int) : bool + existView(viewid : int) : bool + getID(viewid : int) : int + getName(viewid : int) : string + goFirstID() : int + getFirstID() : int + goLastID() : int + getLastID() : int + goNextID() : int + goPrevID() : int + getTexture(viewsid : int, canvasid : int) : Tipus de dades 2::GLuint + getElements(viewsid : int) : ViewsHandler::VectorElements* + getElementsNumber(viewsid : int) : int + getElementsTypeNumber(viewsid : int, type : int) : int + getMinElementsTypeNumber(viewsid : int, type : int) : int + getShaders(viewsid : int) : std::vector< std :: string > + getResolution(viewid : int, rx : int*, ry : int*) + setElement(viewsid : int, el_type : View::View_Elements::Element_Type, indexscene : int) : bool + setActiveCam(viewsid : int, indexscene : int) : bool + setActiveCam(viewsid : int, cam : S2N::S2N_Cam*) : bool + getActiveCam(viewsid : int) : S2N::S2N_Cam* + draw(viewid : int, canvasid : int, tipuscamera : int) + update(viewid : int, upd : bool) + update() + size() : int

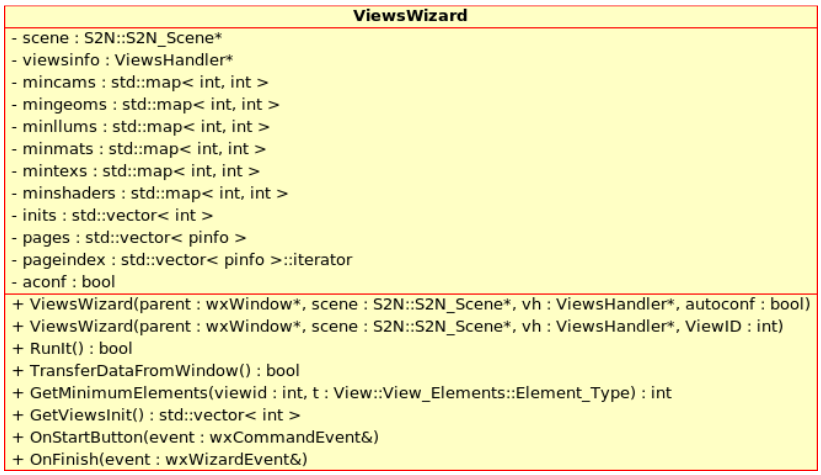
Aquesta classe gestiona els *Views* de l'aplicació, proporcionant els mètodes necessaris per tal d'obtenir informació dels *Views*, recorre'ls, inicialitzar-los, ... és a dir, fa de vincle i proporciona funcionalitats a d'altres classes: **LogicController** la instancia i la usa per extreure'n informació; **ViewsWizard** modifica els elements de l'escena usats per cada *View* fent servir aquesta classe; **ViewFactory** és usada per inicialitzar els *Views*; està íntimament relacionada amb **S2N_Scene**, amb la qual forma el vincle conceptual entre els *Views* i els elements de l'escena que fan servir aquests; manté una relació amb **View_Elements** per seleccionar els tipus d'elements; i finalment aquesta classe està encarregada també de refrescar el contingut dels **GUI_Canvas**, a través de la seva funció **draw()**. Fixem-nos una mica més en aquesta funció:

draw(int viewid**,** int canvasid**,** int tipuscamera**)**



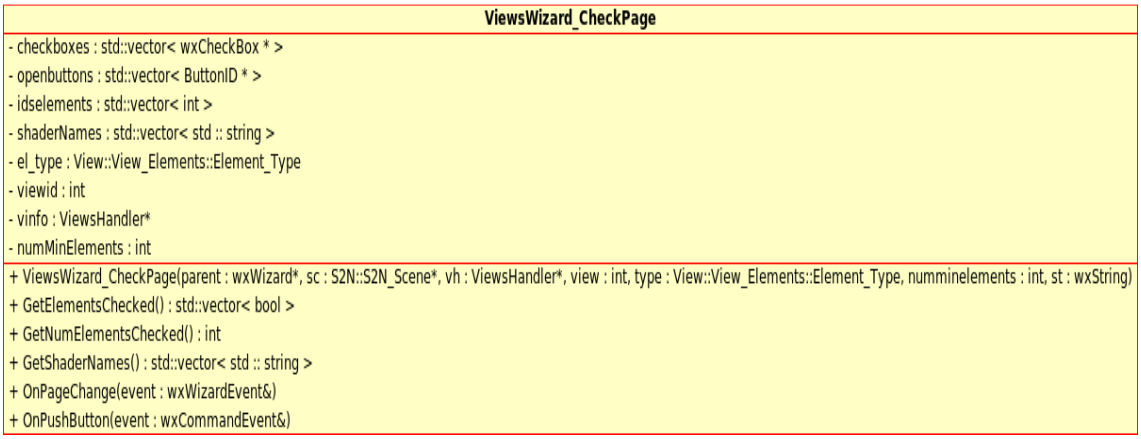
Primer comprovem si l'identificador passat és un *View* inicialitzat i obtenim després la càmera activa d'aquell *View* a través de **S2N_Scene**. A continuació creem una càmera d'aquell tipus copiant-ne els atributs (**newCam**) per després modificar-los segons el tipus de càmera indicat a l'entrada de la funció: si tenim una càmera per mirar des del costat esquerra per exemple, modificarem **newCam** per girar-la 90 graus. Finalment usarem la nova càmera i els paràmetres que tingui el *canvas* indicat per indicar-li al *View* corresponent que repinti la geometria.

ViewsWizard (Main)



Crea un Wizard que deriva de la classe **wxWizard** de **wxWidgets**. El Wizard presenta els elements necessaris per configurar el View i després, pàgina a pàgina ens insta a seleccionar els elements necessaris de cada tipus d'entre els elements de l'escena. Finalment presenta una pàgina on seleccionar la resolució de la textura que generarà el *View* i un botó per iniciar l'operació d'inicialització. Aquesta classe usa **ViewsHandler** per assabentar-se dels Views i també **S2N_Scene** per presentar els noms dels elements de l'escena. Aquesta classe, tot i que formalment hauria d'estar inclòs en el paquet **GUI**, s'ha decidit incloure'l a **Main** per l'intens ús que fa d'elements com **ViewsHandler** i **S2N_Scene**.

ViewsWizard_CheckPage (Main)

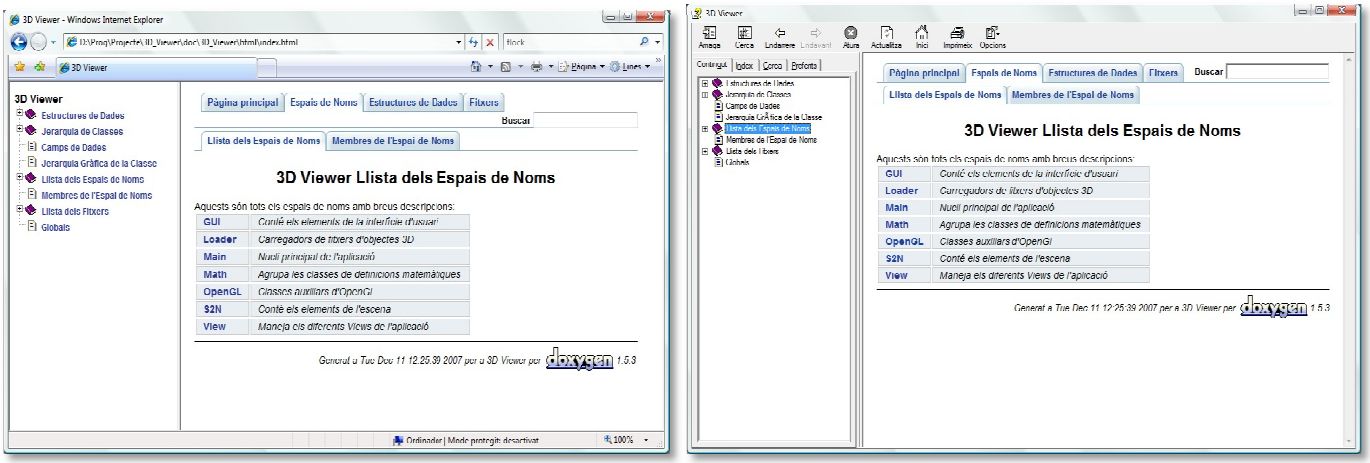


Deriva de la classe **wxWizardSimplePage** de **wxWidgets** i defineix una pàgina del *Wizard* que mostra els elements de l'escena per a poder-los seleccionar i assignar-los als *Views*. **ViewsWizard** usa aquesta classe per crear algunes pàgines del *Wizard.* Aquesta classe usa també **ButtonID** per crear la interfície.

especificació de les classes i els espais de noms

Per tal d'oferir una descripció de la definició de les classes creades, amb les seus mètodes, atributs i el diagrama de col·laboració que té amb altres classes s'ha optat per generar-ho amb **doxygen** (www.doxygen.org), una eina per crear documentació a partir d'ordres inserides com a comentaris dins el codi font.

Per obtenir informació detallada de les classes, es proposa accedir a la documentació citada, disponible en format html o bé en format d'ajuda windows compilada chm, així com a annex de la memòria, en el document "**Memòria – Annex A: Referència de Classes i Fitxers de Codi Font**" que s'adjunta amb el projecte.



EPS
Escola Politècnica Superior
UdG

Projecte/Treball Fi de Carrera

Estudi: Eng. Tècn. Informàtica de Gestió. Pla 2001

Títol:
Entorn de treball per a visualitzar escenes tridimensionals

Document:
Memòria – Annex A:
Referència de Classes i Fitxers de Codi Font

Alumne: Salvi Pi Bonany

Director/Tutor: Mateu Sbert Casasayas

Departament: Informàtica i Matemàtica Aplicada

Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes/any): 01/08

resultats

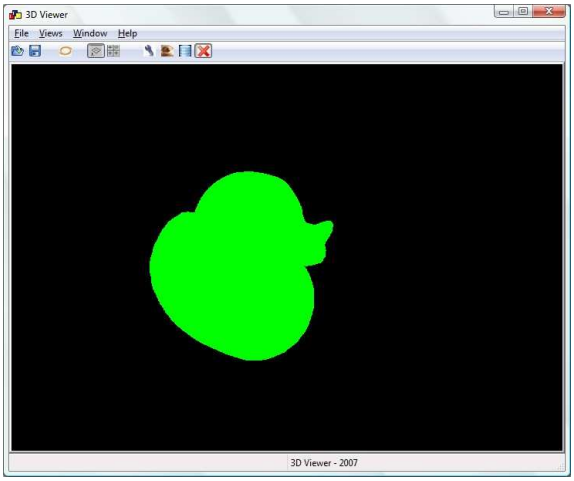
Aquesta secció pretén mostrar els resultats del projecte, que és el desenvolupament de l'aplicació **3D Viewer**. Per un costat, per mostrar la interfície ens podem remetre a la **Guia d'Usuari**, que podem trobar com a annex d'aquesta memòria al document "**Memòria – Annex B: Guia d'Usuari"**.

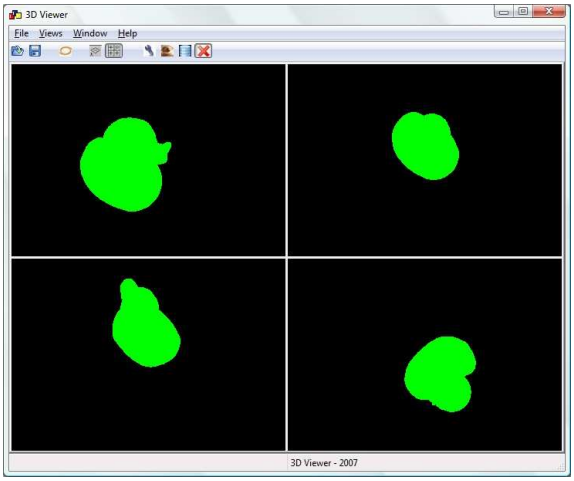
Aquesta aplicació "corre" sota sistemes **Windows**, i per executar alguns dels seus *Views* necessitarem posseir una targeta gràfica que suporti **OpenGL** en la seva versió **2.0**, per tal d'executar els *shaders* que composen aquests *Views*. Tot i que aquesta aplicació està centrada en sistemes **PC Wind**ows i que no s'ha provat en altres plataformes, té la capacitat teòrica de poder-se reconvertir amb pocs canvis a d'altres plataformes. Això és possible per la capacitat multiplataforma de la majoria d'eines que s'han usat, i que s'han explicat en apartats anteriors, com les **wxWidgets**, l'**OpenGL**, les **GLEW** i **COLLADA**. Així podríem, amb relativament poques modificacions del codi font, obtenir versions del programa en plataformes **Unix** o **Mac OS** per exemple.

A continuació es presenten alguns resultats obtinguts usant diferents *Views*, que incorporen diferents tècniques, usant un PC equipat amb Windows XP i una tarja gràfica NVIDIA de la sèrie 6800, amb capacitat OpenGL 2.0.

frame buffer

Aquí hi ha el resultat d'aplicar una de les tècniques més senzilles d'OpenGL: renderitzem la geometria al que s'anomena un FBO, és a dir, un **Frame Buffer Object**, sense tenir en compte ni materials, ni llums, ni textures, només els triangles que formen la figura.

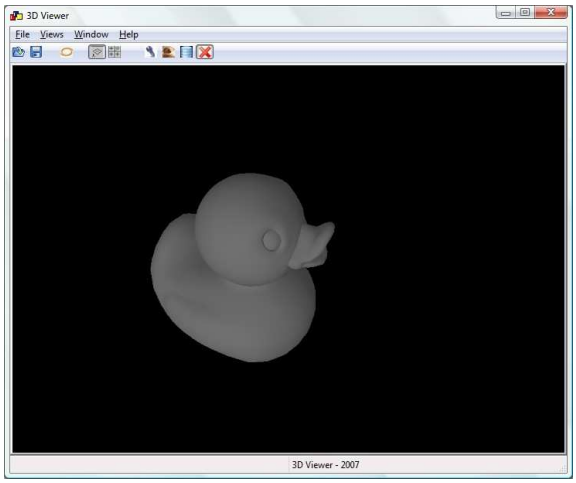


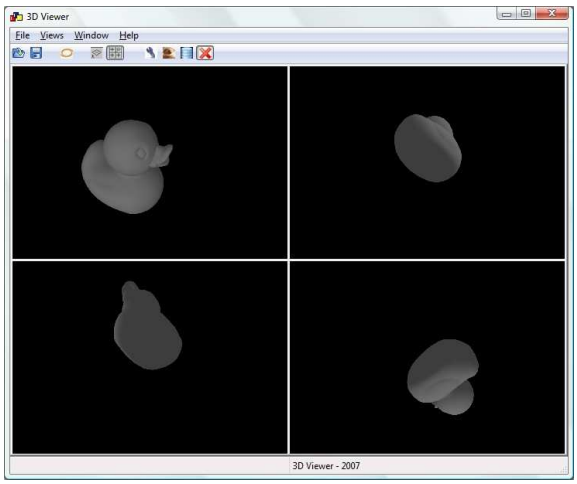


Un FBO és un objecte que representa un **Frame Buffer** (la zona de memòria on pintem el resultat final), de tal manera que podem renderitzar en aquest altre *buffer* sense presentar aquest resultat per pantalla. Quan el volem mostrar tan sols hem de cridar aquest objecte per a que apareixi per pantalla.

frame buffer amb llums

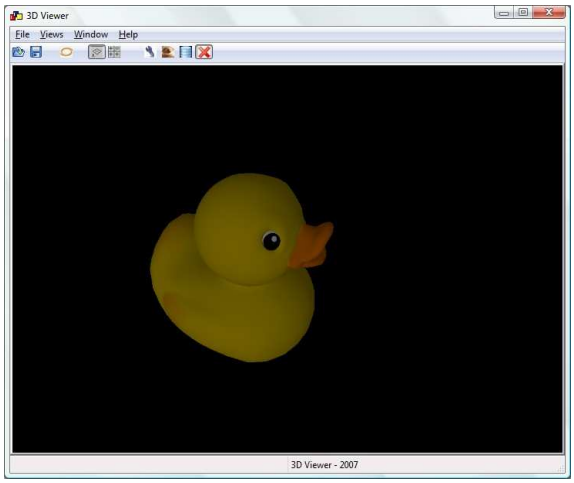
El mateix cas que l'anterior, però tenint en compte les llums i els materials

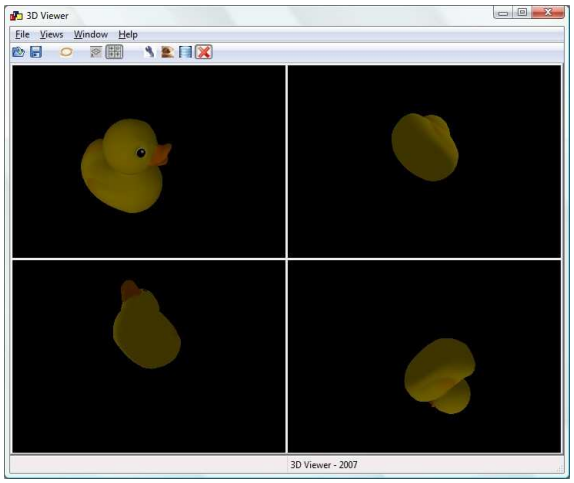




frame buffer amb llums i textura

El mateix que en l'apartat anterior (frame buffer amb llums), però afegint textures a la geometria.





GL Shading Language

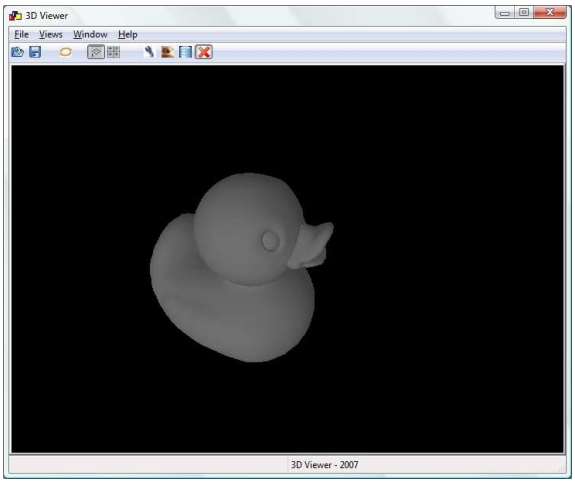
Aquest resultat, que és imperceptiblement igual a l'anterior, està generat no obstant amb un *shader* que implementa la mateixa funcionalitat fixada d'OpenGL amb els llums:

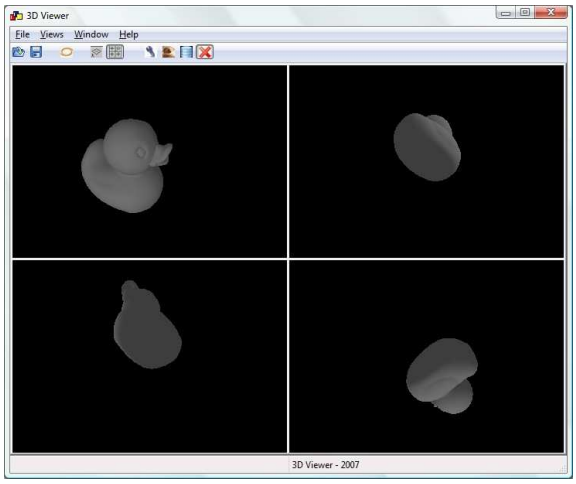
Fragment Shader:

```
void main()
{
	gl_FragColor = gl_Color;
}
```

Vertex Shader:

```
void main()
{
	vec3 normal, lightDir, viewVector, halfVector;
	vec4 diffuse, ambient, globalAmbient, specular = vec4(0.0);
	float NdotL,NdotHV;

	/* first transform the normal into eye space and normalize the result */
	normal = normalize(gl_NormalMatrix * gl_Normal);

	/* now normalize the light's direction. Note that according to the
	OpenGL specification, the light is stored in eye space. Also since
	we're talking about a directional light, the position field is actually
	direction */
	lightDir = normalize(vec3(gl_LightSource[0].position));

	/* compute the cos of the angle between the normal and lights direction.
	The light is directional so the direction is constant for every vertex.
	Since these two are normalized the cosine is the dot product. We also
	need to clamp the result to the [0,1] range. */
	NdotL = max(dot(normal, lightDir), 0.0);

	/* Compute the diffuse, ambient and globalAmbient terms */
	diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
	ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
	globalAmbient = gl_LightModel.ambient * gl_FrontMaterial.ambient;

	/* compute the specular term if NdotL is  larger than zero */
	if (NdotL > 0.0) {
		NdotHV = max(dot(normal, normalize(gl_LightSource[0].halfVector.xyz)),0.0);
		specular = gl_FrontMaterial.specular * gl_LightSource[0].specular *
pow(NdotHV,gl_FrontMaterial.shininess);
	}

	gl_FrontColor = globalAmbient + NdotL * diffuse + ambient + specular;
	gl_Position = ftransform();
}
```





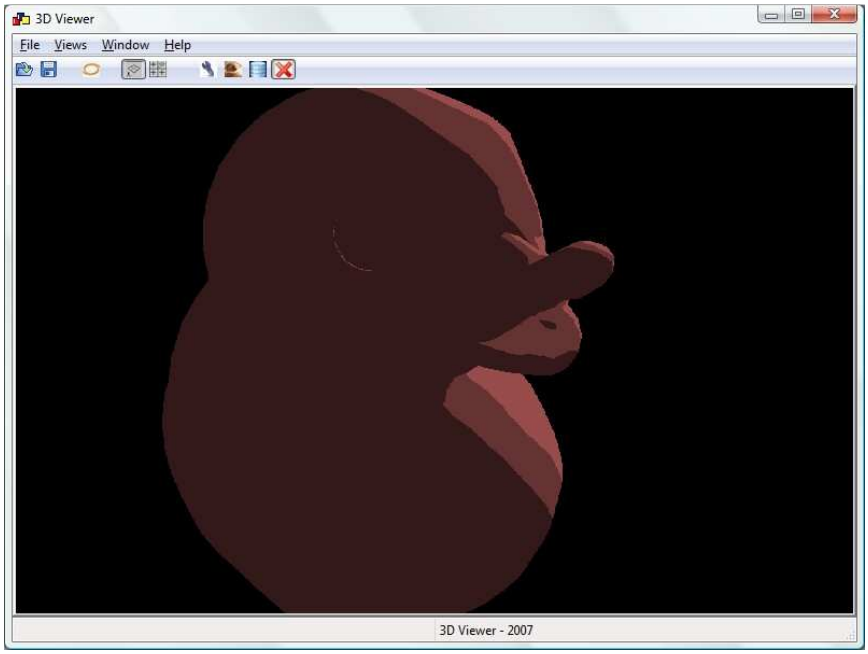
Un altre *shader*, l'anomenat **Toon Shader**, proporciona un colorejat semblant als dibuixos animats, amb aquest codi:

Fragment Shader:

```
varying vec3 lightDir,normal;
varying vec4 lightAmbientColor;

void main()
{
          float intensity;
          vec4 color;

          // normalizing the lights position to be on the safe side

          vec3 n = normalize(normal);

          intensity = dot(lightDir,n);

          //color.x = dot(intensity, lightAmbientColor.x);
          //color.y = dot(intensity, lightAmbientColor.y);
          //color.z = dot(intensity, lightAmbientColor.z);
          //color.a = dot(intensity, lightAmbientColor.a);

          if (intensity > 0.95)
                    color = vec4(1.0,0.5,0.5,1.0);
          else if (intensity > 0.5)
                    color = vec4(0.6,0.3,0.3,1.0);
          else if (intensity > 0.25)
                    color = vec4(0.4,0.2,0.2,1.0);
          else
                    color = vec4(0.2,0.1,0.1,1.0);

          gl_FragColor = color;
}
```

Vertex Shader:

```
varying vec3 lightDir,normal;
varying vec4 lightAmbientColor;

void main()
{
          lightDir = normalize(vec3(gl_LightSource[0].position));
          lightAmbientColor = gl_LightSource[0].ambient;
          normal = gl_NormalMatrix * gl_Normal;

          gl_Position = ftransform();
}
```



Brick GLSL

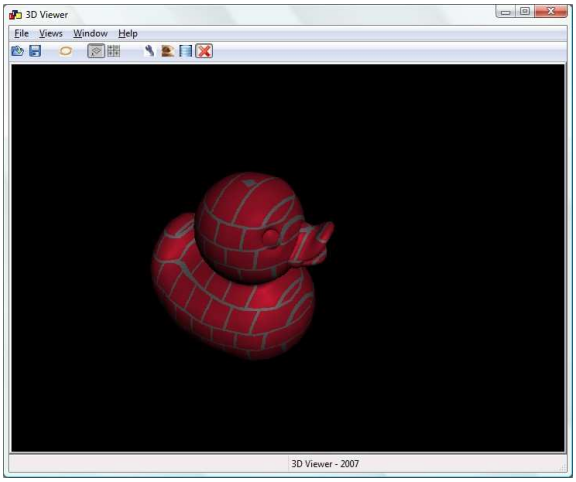
En aquest altre exemple hem creat un nou *View*, a partir del *View* ***GLSL,*** que usa un paràmetre anomenat LightPosition del *shader* per obtenir la posició d'una llum. Veiem-ho al codi OpenGL del *View*:

```
GLuint * GLSLBrick_View::viewinit(int rx, int ry, std::vector<std::string> shadernames)
{
	...

	//Loacalitzacio de les Variables a passar als Shaders
	par[3] = glGetUniformLocationARB(program, "LightPosition");

	...

	//Retornar parametres
	par[0] = tex;
	par[1] = fb;
	par[2] = program;

	return par;
}

void GLSLBrick_View::draw(S2N::S2N_Cam * cam, GLuint * params)
{
	GLuint tex = params[0];
	GLuint fb = params[1];
	GLuint prog = params[2];
	GLuint lightLoc = params[3];

	int num_geos = NUM(S2N::S2N_Geom *);
	S2N::S2N_Geom * geo = GET(S2N::S2N_Geom *,0);
	S2N::S2N_Light * light = GET(S2N::S2N_Light *,0);

	//Get width and height from the texture
	glEnable(GL_TEXTURE_2D);
	glBindTexture(GL_TEXTURE_2D,tex);
	GLint width, height;
	glGetTexLevelParameteriv( GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &width);
	glGetTexLevelParameteriv( GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &height);
	glBindTexture(GL_TEXTURE_2D,0);
	glDisable(GL_TEXTURE_2D);

	glViewport(0,0,width,height); //mida de la textura

	//Bind to FBO: ara pintem a la textura
	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
	{
		glUseProgramObjectARB(prog);

		glPushMatrix();
		glLoadIdentity();

		glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
		glShadeModel(GL_SMOOTH);
		glEnable(GL_CULL_FACE);
		glEnable( GL_DEPTH_TEST );

		cam->apply();
		light->apply();

		Vector4 posLight = light->position();
		glUniform3fARB(lightLoc,posLight.x(),posLight.y(),posLight.z());

		for ( int g = 0; g < num_geos ; g++ )
		{
			S2N::S2N_Geom * geo = GET(S2N::S2N_Geom *,g);
			geo->drawVBOElements();
		}

		glFlush();

		glPopMatrix();
		glUseProgramObjectARB(0);
	}
	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0); //unbind FBO
}
```

El shader que hem aplicat, extret del llibre **OpenGL Shading Language**, i que aplica una capa de rajoles a la geometria, és el següent:

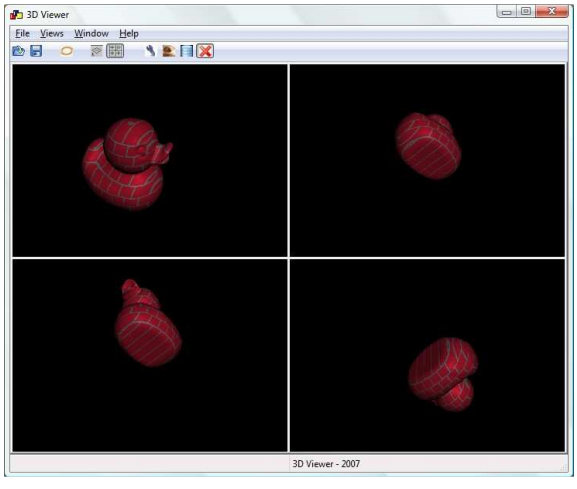
Vertex Shader:

```
const vec3 BrickColor = vec3(0.8,0.1,0.2);
const vec3 MortarColor = vec3(0.5,0.5,0.5);
const vec2 BrickSize = vec2(20,15);
const vec2 BrickPct = vec2(0.9,0.9);


varying vec2 MCposition;
varying float LightIntensity;

void main(void)
{
          vec3 color;
          vec2 position, useBrick;

          position = MCposition / BrickSize;

          if ( fract(position.y * 0.5) > 0.5 )
                    position.x += 0.5;

          position = fract(position);

          useBrick = step(position, BrickPct);

          color = mix(MortarColor, BrickColor, useBrick.x
* useBrick.y);
          color *= LightIntensity;

          gl_FragColor = vec4(color, 1.0);
}
```



Fragment Shader:

```
uniform vec3 LightPosition;

const float SpecularContribution = 0.3;
const float DiffuseContribution = 1.0 - SpecularContribution;

varying float LightIntensity;
varying vec2 MCposition;

void main(void)
{
          vec3 eyePosition = vec3(gl_ModelViewMatrix * gl_Vertex);
          vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
          vec3 lightVec = normalize(LightPosition - eyePosition);
          vec3 reflectVec = reflect(-lightVec, tnorm);
          vec3 viewVec = normalize(-eyePosition);
          float diffuse = max(dot(lightVec, tnorm), 0.0);
          float spec = 0.0;

          if (diffuse > 0.0)
          {
                    spec = max( dot(reflectVec, viewVec), 0.0);
                    spec = pow(spec, 16.0);
          }

          LightIntensity = DiffuseContribution * diffuse + SpecularContribution * spec;

          MCposition = gl_Vertex.xy;
          gl_Position = ftransform();
}
```



conclusions

Partint de l'apartat de resultats anterior podem veure que el món dels gràfics, i més amb la introducció d'eines com GLSL i les actuals targetes gràfiques, té unes possibilitats immenses pel que fa no només a la il·luminació, sinó a altres àmbits, com s'ha esmentat també a la introducció d'aquesta memòria.

És així que en primera instància aquest projecte prengué sentit, com a mètode per programar i provar nous mètodes de visualització, emprant les instruccions adients per explotar les capacitats de les noves generacions de targetes gràfiques.

Així, d'aquesta manera, en el desenvolupament d'aquest projecte, s'han treballat i aprehès el funcionament de moltes eines, mètodes i biblioteques. Partint des del propi **UML**, per crear una estructura primigènia que fes de referent a l'hora de desenvolupar, fins a les biblioteques més especialitzades. S'ha pres consciència del flux de dades que utilitzen les llibreries com **OpenGL** per arribar a visualitzar quelcom a la nostra pantalla. En aquest sentit, s'han estudiat i incorporat les **extensions** d'OpenGL; és a dir, les instruccions que més ràpid incorporen noves funcionalitats de les últimes targes gràfiques. Ens hem servit per això de la biblioteca **GLEW**, que proporciona com hem vist un senzill accés i gestió d'aquestes extensions. En la mateixa línia hi ha també les primeres nocions que s'han captat del **GLSL**, el llenguatge de *shaders* que ens permet programar les targes gràfiques. No és gens menyspreable tampoc els coneixements adquirits en relació al *hardware* gràfic en general, que ens permet cada vegada realitzacions i avenços més sorprenents.

Un altre dels aspectes interessants que ha aportat el desenvolupament de l'aplicació és la construcció d'una interfície gràfica usant **wxWidgets**. Tot i que existeixen algunes eines per crear gràficament les interfícies usant aquest *toolkit* (a l'estil que es fa amb Visual Studio o altres eines de programació visuals per crear interfícies amb botons, finestres, ...), l'aplicació s'ha desenvolupat a partir només de la compilació de codi en C++ que acomplia aquesta funció, sense altres eines. Encara que això ha retardat el desenvolupament de l'aplicació, ha permès per altra banda conèixer força a fons tan el funcionament d'una ***GUI*** *(Graphics User Interface)*, amb els seus elements, el funcionament i gestió de les interrupcions d'usuari, etc., així com controlar i guiar la majoria d'aspectes d'aquest.

Un dels elements més interessants en l'elaboració del projecte ha estat també el món de **COLLADA**. Aquest estàndard, a part de ser obert, té la particularitat d'estar suportat per la majoria d'aplicacions de modelatge tridimensional i d'animació. Això el converteix en el candidat idoni a l'hora de triar-lo per incloure'l a l'aplicació resultant. Els fitxers amb definicions de models en l'estàndard COLLADA poden tenir un nivell de complexitat gens menyspreable, que superava clarament l'abast d'aquest projecte. No obstant, en un primer nivell, l'operativitat que s'ha aconseguit és suficient per als objectius del projecte. Les biblioteques disponibles per gestionar aquest estàndard han estat de gran ajuda en l'elaboració de l'aplicació ***3D Viewer*** resultant.

Al llarg del desenvolupament, s'han investigat diverses alternatives que finalment s'han descartat o usat en menor mesura, però que han deixat alguna idea o saber. Per esmentar-ne només algunes: ***FOX toolkit***, una biblioteca per crear GUIs; **Cg**, un llenguatge a l'estil del GLSL; o bé la definició dels fitxers amb informació de models tridimensionals **OBJ.**

Els resultats presentats a l'apartat anterior són el producte que s'ha obtingut del desenvolupament del projecte: una aplicació que ens permet usar diferents mòduls de visualització (anomenats a la documentació *Views*) per presentar-nos un model tridimensional. Alhora es crea un entorn de treball per mostrar i manipular certs paràmetres dels elements que conformen l'escena, tals com llums, càmeres, materials, ...

Què és, doncs, el més destacable del projecte presentat? Evidentment els coneixements que s'han adquirit en les diverses matèries treballades, i que no són només específiques del camp dels gràfics. Aquest projecte, fins arribar a la consecució d'un resultat presentable, ha requerit moltes hores d'estudi previ, de conceptes al principi un xic abstrusos, que es van clarificant a l'hora de fer proves i obtenir errors. De vegades errors frustrants, ja que aquests no sempre són traçables i predictibles en el món del gràfics. Però també hi ha alegries, com quan es pot obtenir d'una manera tan visual el resultat d'un esforç.

En conclusió, l'aplicació elaborada resultant té una vessant clarament indiscutible: l'aprenentatge que ha suposat en diversos aspectes al seu creador.

treball futur

En aquesta secció vull expressar parts de l'aplicació que serien millorables en properes revisions i versions de l'aplicació.

La part més indicada per introduir noves funcionalitats, és la part de la càrrega d'objectes Collada. Clarament l'establiment d'un nou carregador de Collada milloraria molt la importació de models i altres elements. També es podria implementar l'emmagatzematge d'elements de l'escena a través de l'esquema Collada, en comptes de fer-ho com ara en un fitxer propi de l'aplicació. L'actual implementació del carregador d'escenes, tan sols obté tots els punts de la geometria i els elements de les biblioteques de l'esquema de Collada, impedint l'aplicació correcta dels materials o bé les animacions. Les definicions d'escenes amb elements repetits tampoc és suportada per l'actual versió, ni tampoc els models no triangulats. Com ja s'ha comentat a l'apartat de consideracions de disseny, l'objectiu del projecte no era aconseguir un carregador òptim de Collada, però certament aquesta característica ajudaria a estendre la funcionalitat.

Un altre element que es podria millorar és la visualització de la posició dels llums de l'escena a través de controls gràfics dibuixats a l'escena. Es mostrarien així els llums i es podrien modificar a través d'altres controls, a la manera que s'explora la geometria amb el ratolí. Així mateix els controls que es mostren sobre els llums podrien ser més nombrosos, ja que un llum té més propietats.

Els canvis en el carregador de Collada comportaria segurament la redefinició de les classes que contenen elements de l'escena. També es podria plantejar usar directament les classes proporcionades per la biblioteca **COLLADA DOM** per contenir els elements de l'escena, però aquesta opció faria més difícil usar altres models d'objectes tridimensionals.

Tal com està l'estructura, es podrien introduir nous carregadors per diferents models tridimensionals a més de Collada, com **Maya** o **3D Studio** per exemple.

Un altre objecte millorable seria la gestió dels *shaders*, permetent la càrrega d'altres fitxers des del panell de Shaders. També es podria incloure la possibilitat de manejar variables **uniform** des d'algun control visual, per canviar-ne els valors més fàcilment. Per fer això últim, seria necessari, però, revisar el codi font del *shader* per trobar aquestes variables.

El *View* podria també permetre el canvi d'objectes d'escena sense haver de reiniciar-lo. Es podria desenvolupar per exemple una interfície on pogués arrastrar i deixar anar els elements de l'escena al *View* desitjat.

bibliografia

- *Collada Comunity Web Site*, www.collada.org
- Rémi Arnaud & Mark Barnes, *COLLADA, Sailing the Gulf of 3D Digital Content Creation*, A.K. Peters, Ltd., 2006
- *OpenGL, Official Web Site*, www.opengl.org
- GPGPU.org, www.gpgpu.org
- *wxWidgets, Cross-Plattform GUI Library*, www.wxwidgets.org
    - *First Tutorial*, http://www.wxwidgets.org/docs/tutorials/hello.htm
    - *Tutorials*, http://zetcode.com/tutorials/wxwidgetstutorial/
- *NVIDIA Developer Web Site,* http://developer.nvidia.com/page/opengl.html
- *Visual Studio 2005 MSDN Library Reference,* http://msdn2.microsoft.com/en-us/library/60k1461a(VS.80).aspx
- Julian Smart & Kevin Hock, *Cross-Platform GUI Programming with wxWidgets*, Prentice Hall, 2005
- OpenGL Architecture Review Board, J.Neider, T.Davis, M.Woo, *OpenGL Programming Guide, Fourth Edition: The Official Guide to Learning OpenGL, Version 1.4*, Addison-Wesley, 2004
- Randima Fernando & Mark J. Kilkgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics,* Addison-Wesley, 2005
- Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*, Addison-Wesley, 2003
- Randi J. Rost, *OpenGL Shading Language*, Addison-Wesley, 2004
- *WikiPedia*, *The Free Encyclopedia,* www.wikipedia.org