

Pràctica 2: ZBuffer - Fase 0

GiVD - curs 2016-2017

Objectiu general de la Pràctica 2: Aprendre conceptes bàsics de càmera i il·luminació en visualitzacions projectives (Z-Buffer), programant la targeta gràfica utilitzant *shaders* amb glsl.

FASE 0: Introducció a l'arquitectura d'una aplicació gràfica. Comprovació de la instal·lació dels drivers i primers programes d'exemple.

La pràctica 0 té per objectiu que et familiaritzis amb l'arquitectura d'una aplicació gràfica visualitzant un model poligonal utilitzant i programant amb **OpenGL** i **C++**. En aquest exemple el model concret és un cub.

En aquesta primera fase, no s'ha de lliurar cap codi. Cal provar i testejar els drivers de la targeta gràfica instal·lats i comprendre l'arquitectura que hi ha darrera d'una aplicació gràfica basada en la GPU. Per això també pots llegir els capítols 1.7 i 2.8 del llibre de referència de l'assignatura. Llegeix-los un cop hagi baixat i explorat el codi.

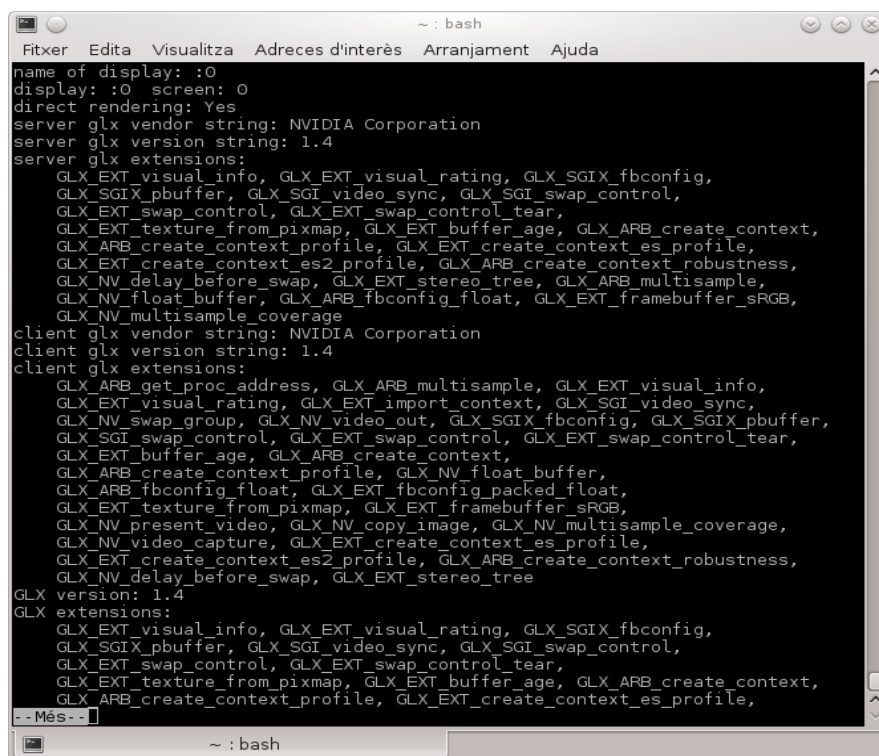
Aquesta fase es compon de 4 passos:

- comprovació de la instal·lació de la targeta gràfica i la instal·lació d'OpenGL,
- visualització d'un model poligonal amb OpenGL
- visualització del model re-programant la GPU (glsl)
- visualització del model utilitzant textures a la GPU (glsl)

Nota: Intenteu respondre les preguntes que es fan a cada apartat per a validar que heu entès tota l'arquitectura de les aplicacions. Al final trobareu un exercici voluntari proposat per a reforçar els conceptes de la pràctica 0.

PAS 1: Comprovació de la instal·lació de la targeta gràfica i d'OpenGL

Instal·lació de OpenGL. Reviseu si teniu instal·lat en el vostre computador les llibreries d'OpenGL. Des de Linux pots executar, des de la consola, la comanda **glxinfo**. Si tens una targeta Nvidia, t'hauria de sortir un missatge com:



```
~ : bash
name of display: :0
display: :0 screen: 0
direct rendering: Yes
server glx vendor string: NVIDIA Corporation
server glx version string: 1.4
server glx extensions:
  GLX_EXT_visual_info, GLX_EXT_visual_rating, GLX_SGIX_fbconfig,
  GLX_SGIX_pbuffer, GLX_SGI_video_sync, GLX_SGI_swap_control,
  GLX_EXT_swap_control, GLX_EXT_swap_control_tear,
  GLX_EXT_texture_from_pixmap, GLX_EXT_buffer_age, GLX_ARB_create_context,
  GLX_ARB_create_context_profile, GLX_EXT_create_context_es_profile,
  GLX_EXT_create_context_es2_profile, GLX_ARB_create_context_robustness,
  GLX_NV_delay_before_swap, GLX_EXT_stereo_tree, GLX_ARB_multisample,
  GLX_NV_float_buffer, GLX_ARB_fbconfig_float, GLX_EXT_framebuffer_sRGB,
  GLX_NV_multisample_coverage
client glx vendor string: NVIDIA Corporation
client glx version string: 1.4
client glx extensions:
  GLX_ARB_get_proc_address, GLX_ARB_multisample, GLX_EXT_visual_info,
  GLX_EXT_visual_rating, GLX_EXT_import_context, GLX_SGI_video_sync,
  GLX_NV_swap_group, GLX_NV_video_out, GLX_SGIX_fbconfig, GLX_SGIX_pbuffer,
  GLX_SGI_swap_control, GLX_EXT_swap_control, GLX_EXT_swap_control_tear,
  GLX_EXT_buffer_age, GLX_ARB_create_context,
  GLX_ARB_create_context_profile, GLX_NV_float_buffer,
  GLX_ARB_fbconfig_float, GLX_EXT_fbconfig_packed_float,
  GLX_EXT_texture_from_pixmap, GLX_EXT_framebuffer_sRGB,
  GLX_NV_present_video, GLX_NV_copy_image, GLX_NV_multisample_coverage,
  GLX_NV_video_capture, GLX_EXT_create_context_es_profile,
  GLX_EXT_create_context_es2_profile, GLX_ARB_create_context_robustness,
  GLX_NV_delay_before_swap, GLX_EXT_stereo_tree
GLX version: 1.4
GLX extensions:
  GLX_EXT_visual_info, GLX_EXT_visual_rating, GLX_SGIX_fbconfig,
  GLX_SGIX_pbuffer, GLX_SGI_video_sync, GLX_SGI_swap_control,
  GLX_EXT_swap_control, GLX_EXT_swap_control_tear,
  GLX_EXT_texture_from_pixmap, GLX_EXT_buffer_age, GLX_ARB_create_context,
  GLX_ARB_create_context_profile, GLX_EXT_create_context_es_profile,
```

Fixa't que el flag de Direct Rendering estigui activat.

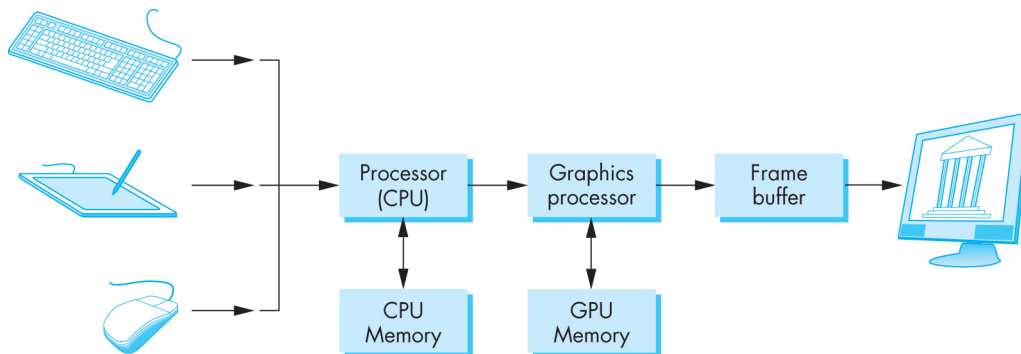
Per a veure les versions instal·lades de GL i dels *shaders*, fes la comanda `glxinfo | grep version` i t'hauria de donar un missatge com:

```
annapuig@ub052043:~$ glxinfo | grep version
server glx version string: 1.4
client glx version string: 1.4
GLX version: 1.4
OpenGL version string: 4.4.0 NVIDIA 340.65
OpenGL shading language version string: 4.40 NVIDIA via Cg compiler
annapuig@ub052043:~$
```

Si no ho tens instal·lat, vés a la plana http://www.opengl.org/wiki/Getting_started on hi han les instal·lacions per a les diferents plataformes i targetes, o consulta la wiki en el Campus Virtual per veure les instal·lacions que fins ara tenim notícia que funcionen.

PAS 2: La primera aplicació GL: visualització d'un model poligonal (cub)

L'arquitectura en la què es basa tota aplicació gràfica és la següent (veure secció 1.1.1 del llibre):



OpenGL és una llibreria que treballa en la memòria de la CPU i encapsula tot l'enviament de dades i gestions a la GPU. El traspàs a la GPU de les dades gràfiques (vèrtexs i colors) pot ser transparent al programador d'OpenGL, només usant crides a OpenGL d'alt nivell. Les aplicacions gràfiques clàssiques funcionen d'aquesta manera.

OpenGL ens ofereix diferents possibilitats per a visualitzar dades. Existeixen diferents funcions que permeten dibuixar punts, rectes, triangles, etc.



Les crides a aquests mètodes (o Function calls) es fan des de la CPU i les dades s'emmagatzemen en la memòria de la CPU. Quan es fa la crida a les funcions d'OpenGL de *display()*, es realitza la transferència a la GPU.

Baixa l'aplicació del campus CubGL.tgz de [l'enllaç del campus virtual](#), descomprimeix-la i obre un nou projecte amb l'IDE QtCreator.

Observeu que en el fitxer .pro (o makefile) hi ha una línia que inclou la utilització de OpenGL. A vegades, si feu un projecte nou, cal editar especialment aquest fitxer per afegir aquesta opció:

QT += opengl

Sense ella, no és possible compilar amb els widgets de gl dins de Qt.

En la versió de Qt5.0 o superior cal incloure també la següent opció:

QT += widgets

Potser, en algunes instal·lacions cal posar en la configuració del qmake del projecte per tal que trobi l'include de gl:

INCLUDEPATH+=/opt/X11/include

Les principals classes d'aquesta aplicació (CubGL) són:

- **main**: conté el programa principal **mainwindow** que conté el disseny de la interfície (hereta de la classe de Qt anomenada QWidget). Es creen els scrollbars i es connecten el seus events (signals) a mètodes concrets que serviran l'event (slots). Podeu trobar més informació del funcionament dels signals i slots en Qt a l'enllaç <http://doc.qt.io/qt-5/signalsandslots.html>
- **glwidget**: conté el widget que permetrà dibuixar amb OpenGL en el seu interior (deriva de QGLWidget). Els mètodes principals a destacar en aquest widget són mètodes que es deriven de la classe abstracte QGLWidget, que són:
 - *initializeGL()*: es crida només el primer cop que s'instancia el widget
 - *paintGL()*: mètode que es crida cada cop que la finestra es refresca, ja sigui per que una altra finestra la tapa, o per què hi ha un canvi de refresc, per exemple per que es canvia un angle de visió.
 - *resizeGL()*: mètode que es crida cada vegada que hi ha un canvi en la mida del widget, per exemple quan s'amplia la finestra.En aquesta classe també es poden sobrecarregar els mètodes de tractament del moviment del ratolí, per exemple.
- **cub**: classe que conté la informació de la geometria 3D, topologia i els colors del cub Qt. D'aquesta classe només cal remarcar, per ara, que és important realitzar el mètode *draw()*, mètode que permetrà pintar tota la geometria en el widget de GL. Els mètodes per a construir la geometria es tractaran més endavant en l'assignatura. Per ara, només ens fixarem que és necessari saber els punts, la seva connexió (o topologia) i els colors associats a cadascun d'ells.

Analitza el funcionament del mètode *draw()*:

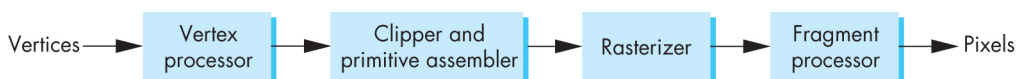
- Des d'on es crida? Per què?
- Dins de quina classe està definit?
- Quina diferència hi ha entre el constructor i el mètode *draw()*? On es defineix la geometria? On es dibuixa? Qui ho envia a la GPU?
- Quina geometria defineix el cub? Les seves cares estan representades per triangles o per quadrilàters?
- Com estan ordenats els punts? Per què?
- Com s'aconsegueix que roti el cub?

En aquest primer exemple no es programa la GPU, però si s'utilitza per la llibreria GL.

PAS 3: Visualització del model re-programant la GPU

En els darrers anys, s'ha obert la possibilitat de treballar i programar directament en la GPU mitjançant el llenguatge GLSL. Això significa la possibilitat de fer la reprogramació directa d'algunes funcions que segueix oferint OpenGL, però tenint constantment la geometria i els colors a la GPU, evitant les transferències de memòria entre els dos processadors (CPU i GPU) i optimitzant així el temps de visualització (veure secció 1.7 i 2.8 del llibre).

Una vegada es tenen els vèrtexs de l'objecte i les seves connexions, es poden realitzar les transformacions geomètriques en el processador de vèrtexs de la GPU. Aquest programa s'anomena *vertex shader*. A més a més, després de rasteritzar els vèrtexs en píxels, es pot programar el càlcul del color de cada píxel en el processador de la GPU de fragments (el programa s'anomena *fragment shader*).



Baixa ara l'aplicació CubGPU.tgz ([de l'enllaç del Campus Virtual](#)), descomprimeix el fitxer i obre un nou projecte amb l'IDE QtCreator. En la versió per a Qt 5.0 s'utilitzen unes estructures auxiliars per a passar informació a la GPU que són els **Vertex Buffer Objects (VBO)**, que són buffers que contenen els vèrtexs de l'Objecte. Amb aquestes estructures permeten passar els vèrtexs i la informació associada a cada vèrtex, com per exemple el color, la normal, etc. per a ser usada en el *vertex shader*.

En aquest punt, es possible que durant l'execució no es trobin els shaders (per exemple en el MAC, segons on tingueu el desplegament del projecte podria ser que haguéssiu de canviar la línia de codi del fitxer `glwidget.cpp`:

```
InitShader( ".././.././../CubGPU/vshader1.glsl", ".././.././../CubGPU/fshader1.glsl" );
```

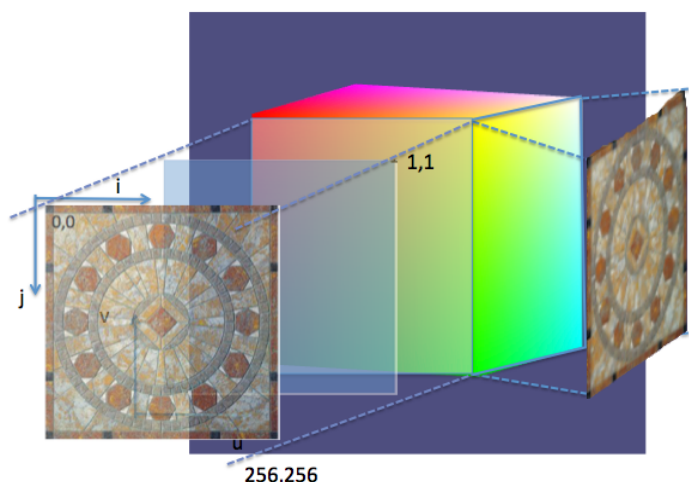
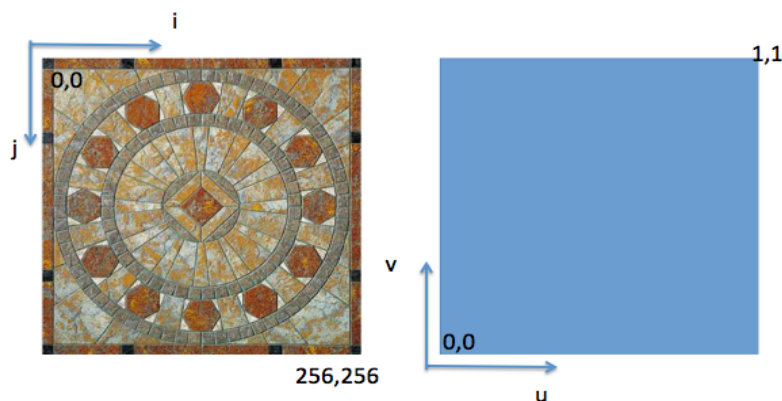
Mira de contestar les següents preguntes:

- Que fa el mètode `draw()`? Des d'un es crida? Per què?
- Dins de quina classe està definit?
- Quina diferència hi ha entre el constructor i el mètode `draw()`?
- On es defineix la geometria? Cada cara del cub està formada per triangles o per quadrilàters? Quin ordre segueixen els punts?
- On es dibuixa?
- Qui ho envia a la GPU?
- Què s'envia a la GPU?
- Quin shader processa els vèrtexs?
- Quin shader s'encarrega dels colors?
- Per què surten els colors purs en els vèrtexs i en les arestes i cares surten degradats?
- Com s'aconsegueix que es roti el cub?

PAS 4: Visualització del model utilitzant textures a la GPU

Fins ara, el model poligonal del cub es visualitza directament amb els colors associats a cadascun dels vèrtexs. En aquest exemple es veurà com es passa una textura a la GPU i s'utilitza per a visualitzar el color a cada píxel. En GL, **les dimensions en píxels de les imatges que formen les textures han de ser potències de 2**.

Disposem de la textura (o imatge `mosaic.png`), tal que cadascun dels seus píxels es mapegen en un espai 2D entre el (0,0) i el (1,1). Aquestes noves coordenades de píxel entre 0 i 1, s'anomenen **coordenades de textura**:



Per a mapejar la textura a cadascun de les cares del cub, cal definir per a cada vèrtex del cub, quin coordenada de textura li correspon.

Així, per a usar una textura cal realitzar els següents passos:

1. Carregar la textura a GL: explora el mètode `InitTextura` de la classe `cub`
2. Cal passar la textura al fragment shader quan es passen les dades a la GPU i lligar la textura amb la variable corresponent del fragment shader.
3. Cal passar les coordenades de textura corresponents a cada vèrtex.
4. Cal activar el mode `GL_TEXTURE_2D` del GL per a que ho activi en el seu *pipeline*.

Baixa ara l'aplicació `CubGPUPTextures.tgz` del [Campus Virtual](#), descomprimeix el fitxer i obre un nou projecte amb l'IDE QtCreator. Fixa't en la classe `cub`. Com ha canviat en relació al projecte anterior?

- Ha canviat el mètode `draw()`?
- On es defineixen les coordenades de textura? Quin mètode les calcula?
- Qui les envia a la GPU?
- Què s'envia a la GPU?
- Quin shader té com a entrada els vèrtexs?
- Quin shader té en compte les coordenades de textura?
- Quin shader utilitza la textura?
- Com canviaries la imatge de la textura mapejada al cub?

Fixa't que la imatge es guarda en una carpeta anomenada **resources** que està integrada en el teu projecte per a tenir independència d'on estan situats els fitxers externs a l'aplicació i fer més portable l'aplicació. Què canvia en relació al projecte `CubGPU`? Com es localitzen els fitxers dels shaders?

Exercici de reforç:

En el projecte `cubGPUPTextures` substitueix el cub per un tetraedre. El tetraedre està format per a quatre vèrtexs $(0,0,1)$, $(0,2\sqrt{2}/3, -1/3)$, $(-\sqrt{6}/3, -\sqrt{2}/3, -1/3)$ i $(\sqrt{6}/3, -\sqrt{2}/3, -1/3)$.

Col·loca una textura a cadascuna de les seves cares. Calcula les coordenades de textura corresponents als seus vèrtexs per a poder situar el dibuix a cada cara.

Quina representació de la malla poligonal és la que estàs utilitzant? Quina seria la més adient en aquest cas (en quan a eficiència de temps i memòria)?