Pràctica 2 – Fase 1

ZBufferToy: Visualització a la GPU

GiVD - curs 2022-2023

Pràctica 2: Fase 1 – Entenent el zBuffer a la GPU

Objectiu de la Fase 1:

L'objectiu d'aquesta fase és doble:

a) Familiaritzar-se amb el codi base: Adaptació de la pràctica 1 a l'algorisme de ZBuffer usant *shaders (o programes executats en la GPU)*,

b) Aprendrem a utilitzar els *shaders* per a programar els models d'il·luminació, de forma que es visualitzarà l'escena amb diferents *shaders* activats en temps d'execució que permetran els diferents tipus d'il·luminació. Per això s'aprendrà a passar diferents valors al *vertex shade*r i al *fragment shader.*

Índex

1. Introducció 1

1.1. Baixa al codi base de la Pràctica 2 1

1.2. Noves classes i adaptació d'algunes de les classes de la Pràctica 1 2

2. Passos a realitzar a la Fase 1 3

PAS 1. Pas de les llums a la GPU 3

PAS 1.1. Pas de la llum ambient global a la GPU 3

PAS 1.2. Pas de la llum de tipus puntual a la GPU. Creació de nous tipus de llums 4

PAS 2. Modificació de la classe Material i pas a la GPU dels valors de materials 6

PAS 3: Implementació dels diferents tipus de *shading (Depth, Normal, Phong-Gouraud, Phong-Phong, BlinnPhong-Gouraud, BlinnPhong-Phong i cel-shading)* 8

3.1. Creació de diferents tipus de shadings 8

PAS 4. Inclusió de les textures en les teves visualitzacions 10

4.1. Inclusió de textures 10

[OPCIONAL] Inclusió mapping indirecta de textures 11

PAS 5. Llegir de fitxers .json de tipus dades geolocalitzades (adaptació del codi de la Pràctica 1) 11

5.1.Llegint fitxers de dades. 11

5.2. Inclusió de textura al Pla base en la lectura de dades geolocalitzades 12

Apèndix A: Exportació de fitxers .obj des de Blender 13

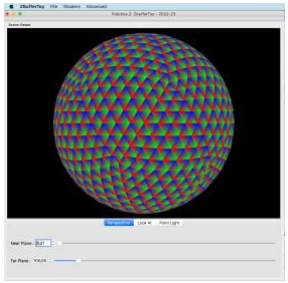
1. Introducció

1.1. Baixa al codi base de la Pràctica 2

- Per a començar a desenvolupar el codi d'aquesta pràctica, es partirà del codi base del classroom del github:

https://classroom.github.com/a/LbGn46PV

- Si l'executes el projecte, inicialment es permet carregar objectes de tipus .obj (es donen objectes a la carpeta "resources/models"). Si carregues per exemple l'objecte sphere0.obj, veuràs que es pinta una esfera de diferents colors. L'esfera està modelada per un model poligonal basat en triangles. Cada triangle es pinta segons els colors associats als seus vèrtexs usant el parell de shaders vshader1-fshader1.



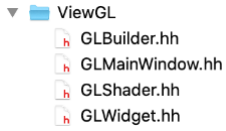
- El codi base està preparat per a canviar la posició de l'observador quan es premi el botó esquerra del ratolí. Si es prem el botó dret es fa zoom.
- Fixa't que tens 3 llengüetes on pots modificar també numèricament algun paràmetre de la càmera i de la llum puntual. Opcionalment, pots afegir més llengüetes per treballar també amb llums direccionals o directament pots estendre el codi per a poder carregar un fitxer de configuració de tipus SetUp que usaves a la pràctica 1.
- Tens diferents fitxers de proves a la carpeta resources, al campus virtual i també en pots baixar-ne d'Internet i generar-ne usant Blender (mira l'apèndix A d'aquest document per veure com exportar .obj des de Blender).

1.2. Noves classes i adaptació d'algunes de les classes de la Pràctica 1

S'utilitzaran algunes classes de la Pràctica 1 amb algunes herències addicionals per a poder-les adaptar a la implementació via GPU. Aconsellem d'anar modificant les classes del projecte base de la Pràctica 2, anomenat **ZBufferToy,** de forma gradual, segons vagis necessitant les funcionalitats que ja tens fetes a la Pràctica 1.

Per a facilitar la integració del codi de la Pràctica 1:

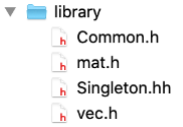
- L'estructura de carpetes és molt similar però s'han afegit tres carpetes que contenen:
  - Les classes que tracten la interacció amb la interfície (ViewGL):



  - Les classes que serveixen per comunicar-se amb la GPU (GPUConnection):

    Totes les classes que necessiten definir els mètodes toGPU() deriven de la classe GPUConnectable. Per exemple, GPUMesh deriva de la classe Mesh per afegir les utilitats de passar les dades a la GPU. En aquesta carpeta també s'han inclòs les factories corresponents que crearan aquests tipus d'objectes i seran les que s'usaran en aquesta pràctica 2.

  - Les classes d'utilitats de matrius i vectors (ara ja no s'usa el glm, sinó el fitxer Common.h que inclou també utilitats més específiques de la càmera i alguna classe addicional que permet definir un template de Singleton usat a les factories) (library):



Si durant el desenvolupament, necessites algunes classes més o canviar alguna signatura d'algun mètode, pots incloure-les en el teu nou projecte.

2. **Passos a realitzar a la Fase 1**

   Aquests són els passos a seguir durant aquesta fase.

   - PAS 1. Pas de les llums a la GPU
   - PAS 2. Creació de les classes GPUMaterial, GPULambertian i GPUMaterialFactory i pas del material a la GPU
   - PAS 3. Implementació dels diferents tipus de *shadings* que es detallen en el menú de Shaders

   - PAS 4. Visualització utilitzant Textures
   - PAS 5. Construcció de l'escena a partir de dades virtuals o dades geolocalitzades (adaptació del codi de la pràctica 1)

A continuació es detalla per a cada pas, una breu descripció, com fer-lo i com comprovar que l'has implementat bé.

PAS 1. Pas de les llums a la GPU

La intensitat ambient global i el vector de llums es troben a la classe GPUSetUp. En aquest apartat cal que aconsegueixis passar i testejar bé el pas de la llum global a la GPU i el pas del vector de llums a la GPU. Per ara, disposes de la classe GPUFactoryLight per crear llum puntuals (GPUPointLigth). Cal també que permetis la creació de llums direccionals i llums de tipus spot-light[1].

PAS 1.1. Pas de la llum ambient global a la GPU
Descripció: Cal que implementis a la classe GPUSetUP, el següent mètode: void GPUSetUP::setAmbientGlobalToGPU(shared_ptr<QGLShaderProgram> program) Com la llum global ambient és igual per a tots els objectes de l'escena, es vol que tots els processadors de la GPU comparteixin els seus valors. Per això cal declarar-la a la GPU com a una variable uniform i des de la CPU, cal indicar-ho també així.
Com fer-ho? 1. Crea el codi per enviar la informació de la llum ambient global des de la CPU a la GPU (mira les explicacions que tens a continuació per saber com passar un vector de tipus uniform). Codifica el vshader1.glsl que tens a la carpeta resources/GPUshaders per a que pugui rebre un vec3 de floats. 2. Prepara el codi per a que es pugui crear la llum ambient global (per ara la pots crear via codi, tot i que podries preparar el codi de GPUSetUP per a llegir-la des de un json). 3. Decideix el moment en el que cal passar la llum ambient global a la GPU (al initializeGL? Al updateGL? En crear un objecte?). Crida el mètode ***setAmbientGlobalToGPU*** des d'allà on creguis convenient.

[1] Material de referencia a la secció 7.2.6 de la web: http://math.hws.edu/graphicsbook/c7/s2.html

a. GPU: Com es defineix una variable de tipus uniform en els shaders? En la **GPU**, en el codi de glsl, per exemple en el vèrtex shader, fixa't en aquesta sintaxi per a definir els valors. Suposa que has de passar un vec4, un vec3 i un float a la GPU, la sintaxi a seguir en glsl és:

```
uniform      vec4 exemple1;
uniform      vec3 exemple2;
uniform     float exemple3;
```

Fixeu-vos que les variables són de tipus "uniform". Això vol dir que serà un valor global i constant per a tots els vèrtex shaders que s'activin en fer el draw().

b. CPU: des del mètode que envia a la GPU: Com es fa el lligam entre les variables de la CPU i la GPU?

Suposant que a la CPU tenim les variables vectorProva (vec4), vector3D (vec3) i unFloat(float), que contenen el valor que volem passar a la GPU, farem el lligam de les variables de la GPU de la següent forma:

```
// 1. En el codi de C++, per a passar els diferents atributs del shader declarem els identificadors
// de les adreces de memòria corresponents a les variables definides a la GPU

    GLuint ex1;
    GLuint  ex2;
    GLuint ex3;

// 2. En el codi de C++, obtenció dels identificadors de la GPU de les posicions de memòria on
//  es contenen les variables
  ex1 = program->uniformLocation("exemple1");
  ex2 = program->uniformLocation("exemple2");
  ex3 = program->uniformLocation("exemple3");
  ...

// 3. En el codi de C++, bind de les zones de memòria que corresponen a la GPU a valors de les variables de la CPU
glUniform4fv(ex1, 1, vectorProva);          // vectorProva és una variable de tipus vec4
glUniform3fv(ex2, 1, vector3D );            // vector3D és una variable de tipus vec3
glUniform1f(ex3, unFloat);                 // unFloat és una variable de tipus GLfloat
```

PAS 1.2. Pas de la llum de tipus puntual a la GPU. Creació de nous tipus de llums

Descripció:

La classe GPULight conté la informació comú a tots els tipus de llums - la intensitat en la què emet (ambient, difusa i especular) i els coeficients d'atenuació en profunditat (constant, lineal i quadràtica) tal com fèieu a la Pràctica 1 -.
Fixa't que la classe GPULight deriva de la classe Light i la classe GPUConnectable (que és qui obliga a codifica el mètode toGPU)

class GPULight: public Light, public GPUConnectable

La classe GPUPointLight deriva d'ella per afegir la informació addicional de la posició. Caldrà que també codifiqui el seu propi mètode toGPU si necessita passar més informació a la GPU.
Finalment, seguint aquest mateix patró, caldrà que codifiquis les noves classes de llum (si no les tenies ja de la pràctica 1), modificant la classe GPULightFactory:

- llums direccionals (direcció)
- llums spot-light (direcció, angle d'obertura)[2]

[2] Material de referencia a la secció 7.2.6 de la web: http://math.hws.edu/graphicsbook/c7/s2.html

En la classe GPUSetUp s'ha afegit un vector de llums i cal implementar un mètode anomenat lightsToGPU() per a que permeti passar la informació de **totes** les llums actives a la GPU, segons la següent capçalera:

```
void GPUSetUP::LightsToGPU (QGLShaderProgram *program)
```

Com fer-ho? Per estructurar la informació de les llums en els *shaders*, s'utilitzarà un "*vector*" per que guardi un "struct" per a cada llum. A continuació es detallen els passos concrets per a passar aconseguir passar un vector d'elements a la GPU, si s'utilitza un "*array*" en els *shaders*.

a. GPU: Per a fer un array on cada element és un struct, anomenat Exemple, en el vèrtex shader podem definir:

```
struct Exemple
{
   vec4 exemple1;
   vec3 exemple2;
   float exemple3;
   ...
};
uniform Exemple conjunt[5];   // En aquest cas serà un array de 5 elements. Aquest valor sempre ha de ser un
                              // numero. NO pot ser una variable!
```

b. CPU: Des del codi de C++, en els mètodes toGPU es fa el lligam de la següent forma:

```
// 1. Es declara un vector d'identificadors de memòria de la GPU
struct gl_IdExemple;
{
      GLuint ex1;
      GLuint  ex2;
      GLuint ex3;
      ....
 }
 gl_IdExemple gl_IdVect [MAX];

// 2. obtenció dels identificadors de la GPU: Suposem i l'index de l'i-èssim element del vector

gl_IdVect[ i ].ex1 =  program->uniformLocation(QString("conjunt[%1]. exemple1").arg( i ));
```

Fixa't que conjunt és el nom de la variable que has definit prèviament en el shader, a la GPU.

```
// 3. Bind de les zones de memòria que corresponen, element a element de l'array
  glUniform4fv(gl_IdVect[ i ].ex1, 1, vectorProva);          // vectorProva és una variable de tipus vec4
```

c. On es declara la llum? Quan es passa a la GPU?

Ara es crea una llum puntual al initializeGL() de la classe GLWidget, quan l'hauries de passar a la GPU? A l'inici de tot? Cada vegada que es visualitza l'escena?

Com comprovo que es passen bé els valors?

1. Comenceu definint només una llum puntual i comproveu que tots els seus atributs es passen correctament a la GPU. Podeu primer posar els valors directament des de codi però també els pots llegir des de la interfície gràfica.
2. Per a validar que passeu bé les dades a la GPU, com no es pot imprimir per pantalla des del *shader*, utilitzeu la variable color del *vertex shader,* com fèieu per la llum ambient global. Si per exemple, li assigneu al color de sortida del *vertex shader*, la intensitat difusa de la llum, hauríeu de visualitzar els objectes del color que heu posat a la intensitat difusa des de la CPU.

3. Definiu ara tres llums puntuals que volen il·luminar l'esfera. Comproveu que es passen bé els seus valors.
4. Definiu altres tipus de llums agafant el codi que teníeu de la Pràctica 1 (direccionals i spot-light). Modifiqueu la classe GULightFactory per a crear els nous tipus de llum. Comproveu que es passen bé els seus paràmetres. En el vector de llums posa una llum de cada tipus i comprova que es passen bé els tres tipus de llums. Què contindrà el "struct" de la GPU? Com l'estructurareu?

PAS 2. Modificació de la classe Material i pas a la GPU dels valors de materials

2.1. Modificació de la classe Material

Descripció:

Aprofiteu la classe Material del codi base de la Pràctica per tal que guardi el **material** d'un objecte. Aquesta classe definirà els atributs de les propietats òptiques d'un objecte (component ambient, component difús, component especular i coeficient de reflexió especular). Cada objecte tindrà associat un material tal i com passava a la Pràctica 1. Els valors del material s'agafaran del fitxer .json que defineixi l'escena.

Cal implementar una nova classe GPUMaterial (seguint el mateix esquema usat a la classe GPULight) per a poder passar els valors de materials a la GPU.

```
void GPUMaterial::toGPU(QGLShaderProgram *program)
```

Utilitzarem també "*structs*" per a estructurar la informació tant a la CPU com a la GPU, tal i com fèiem a les llums. Des d'on es cridarà aquest mètode?

On es declaren els materials?

Cada objecte (o mesh) ha de tenir associat el seu material. Observeu que en la versió de la pràctica base, l'objecte que es carrega té colors que es col·loquen en el vector de colors associats a cada vèrtex per passar-los a la GPU. Ara aquest vector de colors ja no té sentit. Així doncs, cal modificar la classe GPUMesh per a tenir en compte els atributs de material enlloc dels colors i cal que aquests canvis es reflecteixin també en el pas de dades a la GPU en el mètode toGPU.

Per fer les primeres proves podeu directament assignar valors a les diferents components dels materials en el mateix codi, quan es crea la Mesh. Podeu després modificar-ho per a que es llegeixin del .json de l'escena virtual.
Fes un nou parell vèrtex-fragment shader anomenat **vColorShader.glsl** i **fColorShader.glsl**. Codifica el *vertex shader* per a que el pugui rebre convenientment els valors del material que carregues i comprova que les dades estiguin ben passades. Aquest shader hauria de pintar l'objecte segons el seu material difús (més endavant et pot anar bé també per a testejar objectes amb textures)

Com fer-ho?

1. Quan carreguis un .obj, crea i assigna-li un GPUMaterial. En el cas de llegir un fitxer .obj directament, assigna-li un material per defecte. Quan sigui una escena virtual, cal que el llegeixis del fitxer .json, tal i com feies a la Pràctica 1, aquest cop però des de la GPUSceneVirtualFactory.
2. Cal afegir la nova classe GPUMaterial amb el seu mètode toGPU. Mira els punts (a) i (b) que s'explica a continuació per a saber com fer el pas de la informació de la CPU a la GPU
3. Cal que passis aquest material a la GPU quan enviïs la informació de l'objecte a la GPU

Per estructurar la informació en de totes les variables de comunicació dels diferents atributs dels materials, en el codi de C++ i en els *shaders*, s'utilitzarà un "*struct*". Recordeu que per a comunicar la CPU amb la GPU s'han de fer diferents passos: (a) A la GPU cal definir quines variables es volen rebre i (b) A la CPU cal enviar-les. A continuació es detallen els passos concrets per dades a la GPU, si s'utilitza un "*struct*" en els *shaders*.

a. GPU: Com es defineix i s'utilitza un struct en els shaders?

En la **GPU**, en el codi de glsl, per exemple en el vèrtex shader, fixa't en aquesta sintaxi per a definir els valors. Suposa que has de passar un vec4, un vec3 i un float a la GPU, i els volem encapsular en un struct anomenat Exemple. La sintaxi a seguir en glsl és:

```
struct Exemple
{
   vec4 exemple1;
   vec3 exemple2;
   float exemple3;
   ...
};
uniform Exemple test;
```

Fixeu-vos que es la variable **test** és de tipus "uniform". Això vol dir que serà un valor global i constant per a tots els vèrtex shaders que s'activin en fer el draw().

b. CPU: des del toGPU(): Com es fa el lligam entre les variables de la CPU i la GPU?

Suposant que a la CPU tenim les variables vectorProva (vec4), vector3D (vec3) i unFloat(float) i contenen el valor que volem passar a la GPU, farem el lligam de les variables de la GPU de la següent forma:

```
 // 1. En el codi de C++, per a passar els diferents atributs del shader declarem una estructura amb els
identificadors de les adreces de memòria corresponents a les variables definides a la GPU
   struct {
      GLuint ex1;
      GLuint  ex2;
      GLuint ex3;
      ....
   } gl_IdExemple;

  // 2. En el codi de C++, obtenció dels identificadors de la GPU
   gl_IdExemple.ex1 = program->uniformLocation("test.exemple1");
   gl_IdExemple.ex2 = program->uniformLocation("test.exemple2");
   gl_IdExemple.ex3 = program->uniformLocation("test.exemple3");
   ...

  // 3. En el codi de C++, bind de les zones de memòria que corresponen a la GPU a valors de les variables de la CPU
  glUniform4fv(gl_IdExemple.ex1, 1, vectorProva);          // vectorProva és una variable de tipus vec4
  glUniform3fv(gl_IdExemple.ex2, 1, vector3D );            // vector3D és una variable de tipus vec3
  glUniform1f( gl_IdExemple.ex3, unFloat);                 // unFloat és una variable de tipus GLfloat
```

Com es tenen diferents parells de shaders? Com s'activen i desactiven?

a. S'inicialitzen tots els parells de shaders amb crides successives a initShader(..). Ara hauràs de fer un nou parell vèrtex-fragment shader (Color), que es podrà activar des del menú Shaders de la GUI. Mira des de quin mètode caldria activar aquest nou parell de shaders.

b. Per a canviar de shader en la GPU cal linkar i fer bind del nou parell de shaders (mira el codi de la classe GLShader de la carpeta ViewGL)

```
program->link();   // muntatge del shader en el pipeline grafic per a ser usat
program->bind();   // activació del shader al pipeline gràfic
```

c. Estructura els programes que necessitaràs per anar activant i desactivant. Quan canvies de *shader*, hauràs de tornar a passar totes les dades a la GPU.

Si vols utilitzar diferents *shaders* en temps d'execució raona on s'inicialitzaran els *shaders* i com controlar quin *shader* s'usa? Cal tornar a passar l'escena a la GPU quan es canvia de shader?

Quines proves pots fer per saber que funciona?

1. Adapta el codi de GLWidget per a que puguis carregar un nou parell de vèrtex-fragment shader des de l'initializeGL()
2. Posa des de codi només la component ambient del material de l'objecte a color vermell
3. En el *vertex shader* dóna com a color de sortida (out) el valors de la component ambient que estàs passant a la GPU.
4. Comprova que l'objecte es visualitza en vermell.
5. Fes el mateix amb la resta de propietats òptiques (una a una) per a comprovar que estan ben passades a la GPU.
6. Un cop això et funcioni, tracta de llegir una escena virtual i inicialitzar els valors del material del fitxer .json, portant el codi que usaves a la Pràctica 1. Ara es llegeix des de la classe GPUSceneFactoryVirtual. Recorda que Material allà derivava de la classe Serializable.
7. Prova a posar a la teva escena dos objectes amb materials diferents. Es pinta cadascun amb el seu material?

[**Opcional**] pots fer que els valors del material es llegeixin juntament amb un fitxer .obj (en una línia del fitxer que comença amb mtlib es detalla el fitxer que conté el material).

Pots usar els valors que surten en el fitxer sphere0.mtl per a posar valors al material del teu objecte. Trobaràs més detalls del format a les especificacions sintetitzades del format .obj[3] i més detallat a les especificacions completes[4]

PAS 3: Implementació dels diferents tipus de *shading (Depth, Normal, Phong-Gouraud, Phong-Phong, BlinnPhong-Gouraud, BlinnPhong-Phong i cel-shading)*

3.1. Creació de diferents tipus de shadings

Descripció:

En aquest apartat es tracta d'implementar els següents diferents tipus de *shading,* implementant un parell de vèrtex-fragment shader per a cadascun del diferents tipus, si ho creus convenient:

- *Depth shading* (tonalitats grises de profunditat)
- *Normal shading* (colors segons les normals)
- *Gouraud* (suavització del color): versió Phong i versió Blinn-Phong
- *Phong shading* (suavització de les normals): versió Phong i versió Blinn-Phong
- *Cel o Toon Shading* (tècnica no realista)

Tingueu en compte que les variables de tipus **out** del *vertex shader* són les que es rebran com a **in** en el fragment shader, ja interpolades en el píxel corresponent.

Es vol activar cada *shader* via menú (ja ho teniu preparat a la classe GLMainWindow). En la pràctica trobaràs els slots o mètodes que els serveixen estan buits en la classe GLWidget. També ja estan predefinits els shortcuts en la interfície: la tecla Alt+0 activarà el *Color,* la tecla Alt+1 el *Normal Shading, etc.*.

[3] L'especificació completa del format .obj incloent el format .mtl es pot trobar a: https://en.wikipedia.org/wiki/Wavefront_.obj_file

[4] L'especificació completa del fitxer .mtl es pot trobar a: https://www.fileformat.info/format/material/

Breu explicació de cada shader:

Depth-shading (suavització del color segons la profunditat del píxel a pintar): En el *Depth shading* es pinta en grisos la profunditat del triangle que s'ha rasteritzat en el píxel. Dependrà del plans de *clipping* anteriors i posteriors per a poder graduar el nivell de gris de forma correcta.

Normal-shading (suavització del color segons la normal a cada vèrtex): En el *Color shading* es pinta en colors la normal associada a cada vèrtex de la malla i es degraden els colors en els píxels interiors als triangles segons el valor de les normals dels seus vèrtexs.

Gouraud (suavització del color a partir de les normals calculades a cada vèrtex): En el *Gouraud shading* es calcula la fórmula de Phong o la de Blinn-Phong tenint en compte les normals a cada vèrtex i s'interpola el color a nivell de píxels. Fixa't que quan es llegeix un objecte, cada vèrtex ja té la seva normal. Com serà aquest valor de la normal? Uniform o no uniform?

També necessitaràs la posició de l'observador a la GPU per a calcular el vector H del model Blinn-Phong o el vector reflectit en el cas de Phong. Passa la posició de l'observador de la classe càmera a la GPU per a poder fer el càlcul del vector H. En la classe Camera utilitza el mètode **toGPU** per a passar l'observador als *shaders* per a que es passi la posició de l'observador cada vegada que s'actualitza la posició de la càmera amb el ratolí. Com serà aquesta variable al *shader*? Uniform? O IN?

Phong Shading (suavització de les normals a nivell de píxels): En el *Phong Shading* les normals s'interpolen a nivell de píxels. Raoneu on és calcula la il·luminació i modifiqueu convenient els fitxers de la pràctica.

Si vols utilitzar diferents *shaders* en temps d'execució raona on s'inicialitzaran els *shaders* i com controlar quin *shader* s'usa? Cal tornar a passar l'escena a la GPU quan es canvia de shader? I també la càmera?

Cel o toon Shading: *Shading* discret, estil *cartoon*, calculat a cada píxel segons el producte escalar del vector de la llum direccional amb la normal al punt. Es tabula el resultat del producte escalar entre el vector de la llum i la Normal al punt en diferents intervals: [0 ..v1), [v1 .. v2), [v2 .. v3) i [v3 .. 1.0). A cada interval s'assigna un cert color difús corresponent a diferents tonalitats d'un mateix color (color1, color2, color3, color4). Calculeu a nivell de píxel i feu aquest càlcul pel color difús.

Com fer-ho? Passos a seguir:

1. Implementa inicialment el *Depth shading* tenint en compte que en el fragment shader pots accedir a una variable anomenada gl_FragCoord, que conté a la coordenada z, un valor entre 0 i 1. La z corresponent a 0 és la del pla de *clipping* anterior (Near Plane a la interfície) i la de valor 1 es correspon al pla de *clipping* posterior (Far Plane a la interfície). Mira la imatge obtinguda amb el fitxer f16.obj. Fixa't especialment en els valors dels plans de retallat.

2. Implementa després el *Normal shading* amb per aplicar-lo a l'esfera. Per implementar el *Normal shading* de cada objecte cal tenir definida a cada vèrtex la seva normal ponderada segons les normals dels triangles que

confluexen en aquell vèrtex. Per això, a cada vèrtex cal calcular la seva normal segons les cares a les que pertanyi. En els fitxers .obj trobaràs ja les normals calculades a nivell de vèrtex (les línies que tenen prefix vn). Cal passar aquestes normals a la GPU, modificant els mètodes toGPU i draw dels objectes i també el *vertex shader* o el *fragment shader* per a que les tracti a la GPU.

3. Implementa després el *Gouraud shading* amb una llum puntual per aplicar-lo a l'esfera. Volem disposar de dues versions del càlcul de la il·luminació: una basada de Phong i l'altra basada en Blinn-Phong. Cal tenir un parell de vèrtex-fragment shader? O dos? Fixa't que en aquest/s shader/s cal que hi passis tota la informació per a poder calcular la fórmula de Phong o bé la de Blinn-Phong, modificant els mètodes toGPU i draw dels objectes i també el *vertex shader* o el *fragment shader* per a que les tracti a la GPU.
4. Prova amb diferents materials canviant les diferents components de les propietats òptiques.
5. Implementa el *Phong-shading* per a les dues versions de la fórmula de càlcul de la il·luminació. Quina diferència hi ha amb el *Gouraud-shading*? On l'has de codificar? Necessites uns nous *vertex-shader* i *fragment-shader*? Raona on es calcula la il·luminació i modifica convenientment els fitxers de la pràctica.
6. Després implementa l'atenuació amb profunditat a cadascun dels mètodes.
7. Implementa finalment el *Cel* o *Toon-Shading (explicada a teoria).* On s'implementarà el càlcul del color per a tenir més trencament entre las games de colors? Necessites uns nous *vertex-shader* i *fragment-shader*? Raona on es calcula la il·luminació i modifica convenientment els fitxers de la pràctica.

[OPCIONAL] Èmfasi de siluetes

Descripció:

En aquest apartat es tracta de visualitzar els objectes visualitzats amb *toon-shading* afegint-li l'èmfasi de les siluetes. Emfatitzar les siluetes suposa més èmfasi en els contorns de les projeccions 2D dels objectes que s'estan visualitzant. Per això és necessari detectar els píxels que estan en aquests contorns.

Com fer-ho?

Per fer l'efecte de remarcat de les siluetes 2D dels objectes, tenen color més intens el píxels tals que l'angle entre la normal i la direcció de visió sigui diferent de 0. El color del píxel es calcula directament amb la component difusa del material de l'objecte multiplicada per (1- cos (α)), sent α l'angle entre el vector de visió i la normal associada al píxel. Implementa aquest efecte als tres *shaders* que has implementat en el pas anterior.

Pots fer les ampliacions suggerides per al càlcul de siluetes suggerides a la Pràctica 1.

PAS 4. Inclusió de les textures en les teves visualitzacions

4.1. Inclusió de textures

Descripció:

S'inclouran textures en la visualització del objecte que s'ha carregat. Recorda que des de blender s'exporten les coordenades de textura d'una malla poligonal. Amb aquestes coordenades passades al *fragment shader* es poden obtenir els píxels associats a cada píxel on es rasteritza l'objecte.

Com fer-ho?

Modifica la classe GPUMesh per a que pugui tractar la textura que hi té definida ara en el codi base. Considera també que la GPUMesh ha de tenir també els vèrtexs de textura associats a cada vèrtex de l'objecte. Aquests es poden llegir dels fitxers .obj que tenen línies que comencen amb el prefix vt.

```
shared_ptr<QOpenGLTexture> texture;
```

Carrega una textura per l'objecte en la seva constructora (consulta el projecte CubGPUTextures per veure com es feia

a la classe cub). Recorda que la textura sempre s'acaba consultant en el fragment *shader*.

Implementa el mètode de la classe GLMesh:

```
void GLMesh::initTextura()
```

Recorda modificar els mètodes toGPU i draw de l'objecte per a passar els vèrtexs de textura a la GPU, en el cas que l'objecte tingui textura.

Recorda també que la textura es pren com a la component difusa del material o es fa una ponderació amb el material base de l'objecte. *Per exemple, un 75% del color final potser de la textura i un 25% el color del material base.*

[OPCIONAL] Inclusió mapping indirecta de textures

Descripció:

Ara la textura embolcalla l'objecte en un mapping indirecta esfèric. Modifica la classe objecte per a que pugui tenir opcionalment textures directes o textures amb mapeig indirecte.

Suposem inicialment que l'objecte que es llegeix de fitxer està centrat en el punt (0,0,0). Es pot pensar que la textura embolicarà l'objecte com si l'objecte estigués dins d'una esfera (*indirect texture mapping*).

Per a generar les coordenades de textures associades a cada vèrtex **p** del objecte, es considera el vector unitari que va des del centre, **c**, de la capsa mínima contenidora de l'objecte, fins a **p**. Aquest vector serveix per a considerar el punt d'una esfera centrada en **c** i que embolica l'objecte. Les coordenades de textura **(u, v)** es calculen a partir de les coordenades esfèriques de l'esfera seguint la següent fórmula:

$u = 0.5 - \arctan2(z, x)/ 2\pi$

$v = 0.5 – \arcsin(y)/ \pi$

Recorda que (u,v) son valors entre 0 i 1.

NOTA: La funció atan2 s'utilitza en C++, però en glsl, per utilitzar-la en els shaders, la funció és atan

PAS 5. Llegir de fitxers .json de tipus dades geolocalitzades (adaptació del codi de la Pràctica 1)

5.1.Llegint fitxers de dades.

Descripció:

Intenta carregar fitxers .json de tipus REALDATA. Per això caldrà modificar la classe GLBuilder per a poder incloure les modificacions que estan comentades en el codi i permetre que els gizmos siguin directament el fitxer GPUMesh que s'indiqui en la interfície gràfica. En el codi que es proporciona no es fa ni l'escalat ni el posicionament dels objectes que representen les dades.

Porta el codi que ja et funcionava de la Pràctica 1 per a que et permeti posar gizmos de tipus GPUMesh. En el cas que vulguis gizmos de tipus esfera, carregaràs el fitxer .obj que tens en el campus, per exemple, sphere0.obj, i el situaràs i escalaràs segons el valor de propietat. No tinguis en compte tampoc els materials per ara.

Com fer-ho?

Els menús de l'aplicació (del menú Fitxers) ja estan connectats des de la classe GLMainWindow a la classe GLBuilder que serà qui preguntarà pel fitxer a carregar i delegarà la feina de construir l'escena.

Modifica la classe GLBuilder per a que es puguin carregar escenes de dades. En el cas de la lectura de REALDATA, des del GPUSceneFactoyData, quan es crea el pla afitat caldrà crear-li dos triangles que formin el pla. Per això cal que creis una classe FittedPlane (i la seva corresponent GPUFittedPlane) que tingui només una cara acotada definida per 2 triangles (fixa't en el codi d'exemple CubGPU del campus, a la seva classe cub.cpp per a reproduir-lo a la classe GPUFittedPlane però només creant una cara).

5.2. Inclusió de textura al Pla base en la lectura de dades geolocalitzades

Descripció:

Recorda que el pla que has creat en el Pas 1 necessita coordenades de textura per poder-li aplicar la textura correctament. Així, aquest pla ha de tenir associada una textura i els vèrtexs de textura associats a cada punt de l'objecte.

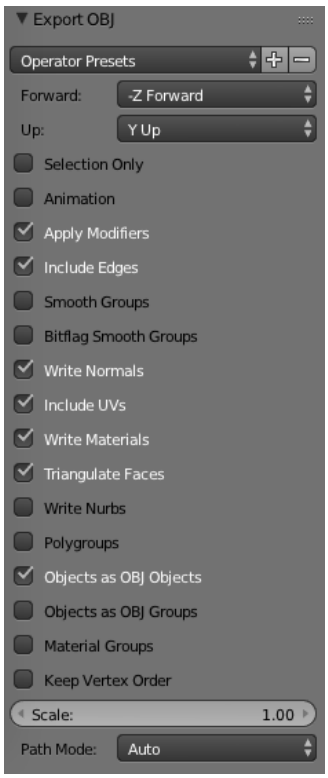
Un cop hagis fet correctament el mapeig de textures, carrega una textura pel terra des de la lectura del fitxer .json. Recorda que la textura sempre s'acaba consultant en el *fragment shader*.

Si no vols tenir els gizmos texturats, hauràs de activar el *shader* que tingui en compte la textura pel pla i activar un *shader* sense textura en la resta d'objectes.

[Opcional] Pots modificar el terra, ja sigui una esfera o un pla amb diferents textures, segons la distància a la que es estigui l'observador, per atenuar més o menys el seu color.

Apèndix A: Exportació de fitxers .obj des de Blender[5]

Per a fer les proves disposeu d'una esfera en el fitxer sphere0.obj, encara que podeu carregar d'altres fitxers .obj continguts en la carpeta resources del projecte o bé de la carpeta anomenada models i textures en el campus virtual (https://campusvirtual.ub.edu/mod/folder/view.php?id=3324987) o bé generats amb blender. Per a generar-los bé des de blender heu d'exportar-los a .obj i seguir els paràmetres següents del menú d'exportació per a generar un fitxer .obj correcte a punt de ser llegit per la Pràctica 2. Fixeu-vos que si heu posat una textura a l'objecte amb blender, també podeu exportar les coordenades de textura.



[5] https://www.blender.org