Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Generació procedural de vegetació en GPU

Document: Memòria

Alumne: Aleix Rius Fabregó

Tutor: Gustavo Patow
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: SLI

Convocatòria (mes/any): Juny 2019

Universitat de Girona
Escola Politècnica Superior

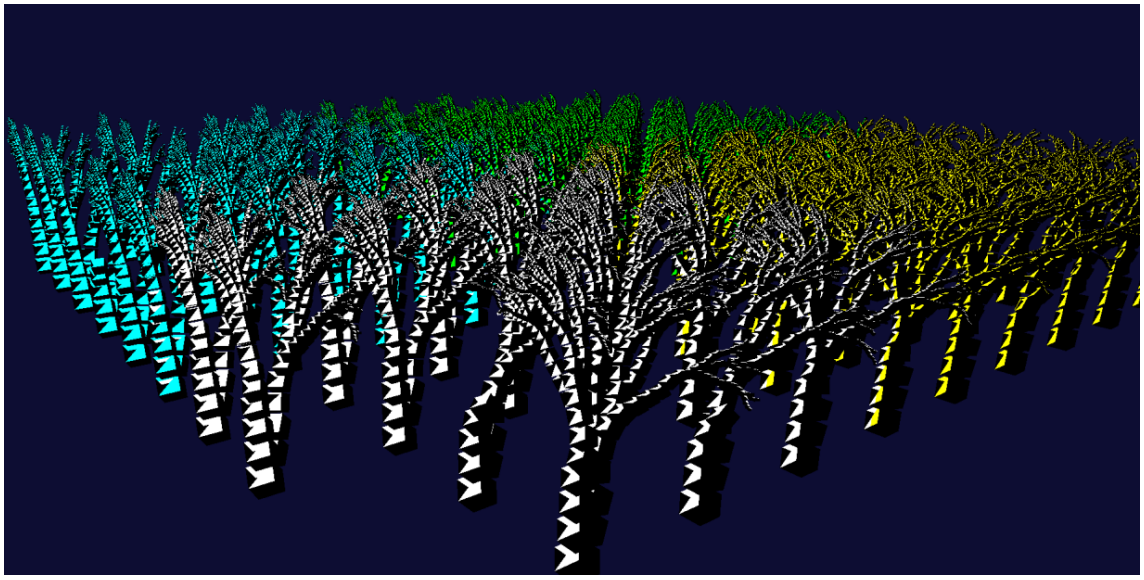
Escola Politècnica Superior

Universitat de Girona

Generació procedural de vegetació en GPU

Treball final de grau

Grau en Disseny i Desenvolupament de Videojocs



Document: Memòria

Alumne: Aleix Rius Fabregó

Tutor: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: SLI

Convcoatòria (mes / any) : Juny 2019

1. Introducció

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

El L-System és una tècnica procedural que es basa amb la reescriptura de gramàtica basat amb regles i símbols. Les aplicacions que té són molt diverses, en aquest projecte ens centrarem amb la generació de vegetació.

1.1 Motivacions

Per norma general, normalment en un treball, recerca, estudi, etc., hi ha dos tipus de motivacions amagades al darrera: les personals i les necessitats professionals. Les personals solen estar més encarades en les ganes de treballar o investigar en un tema en concret. En el meu cas aquesta motivació apareix fruit de la curiositat i la intriga al voltant dels continguts (vegetació, clima, atmosfera, il·luminació, so, ...) d'alguns videojocs, on sense fer-se'n gaire ressò, aconsegueixen uns resultats increïbles. Hi ha molts exemples on tot, o gairebé tot l'escenari és fruit d'una unió entre programadors i artistes que utilitzen tècniques procedurals com a intermediari. Un d'ells és el *Horizon Zero Dawn* [1] un joc que han aconseguit eines tant potents com generar continguts a temps real, on tot i ser procedurals, l'artista té en tot moment el control del que està generant.

Per altra banda també hi ha l'interès per aprendre i aprofundir en la programació de gràfics utilitzant la targeta gràfica, ja que considero que és un coneixement molt valuós dins el camp

[1] Joc de PlayStation 4 https://www.guerrilla-games.com/play/horizon

dels videojocs i que no es veu suficient durant el grau, i per tant, aquest treball serveix per adquirir nous coneixements i posar-los en pràctica.

Un bon començament per aprendre aquests continguts i també entrar al món procedural és utilitzant els L-Systems per generar vegetació, que són tècniques molt conegudes i senzilles, però sempre des de la CPU, en canvi, hi ha molt poca informació de la implementació d'aquests algorismes en GPU.

1.2 Propòsit

Donat que hi ha una manca important d'informació respecte els L-System en targes gràfiques, el propòsit principal és aconseguir una implementació d'un L-System dins una tarja gràfica. A més a més, aprofitat les capacitats de les targes gràfiques, un segon propòsit és aconseguir generar diferents arbres amb temps molt baixos de càlcul. A causa de que no hi ha informació sobre implementació de L-System en les targes gràfiques, en excepció del paper de Parallel Generation of L-System (Markus Lipp, 2009), el qual està pensar i dissenyat per només fer un arbre, aleshores ens trobem dins d'un camp desconegut i amb falta de referents.

1.3 Objectius

L'objectiu principal d'aquest treball és adquirir coneixement i experiència en el desenvolupament de programes per targes gràfiques. Així com conèixer les seves limitacions, avantatges i inconvenients des d'un punt de vista pràctic.

El segon objectiu, no menys important que el primer, és desenvolupar un programa que permeti generar diferents arbres, i que la seva part de processat sigui la GPU i en conseqüència desenvolupar un algorisme nou adaptat per aquestes circumstàncies.

Com a objectius secundaris el programa permetrà generar un nombre d'arbres amb baixos temps de generació, sempre tenint en compte les limitacions de hardware de l'usuari, i aconseguir visualitzar els resultats de forma gràfica i entenedora.

Per aconseguir els objectius, es treballarà des de zero, sense utilitzar cap motor gràfic o de visualització, així com tampoc s'utilitzarà cap software de generació procedural.

Així doncs, el projecte serà una combinació de programació de *shaders* empotrat la llibreria OpenGL i programació convencional usant C++.

Amb tot això en ment, les tasques principals d'aquest treball són:

- Aprendre el funcionament de tècniques procedurals de generació de vegetació (L-System)
- Generar una aproximació al modelat de vegetació emprant la CPU.
- Estudi de la llibreria d'OpenGL.
- Definició de l'estructura de dades per optimitzar el rendiment.
- Implementació de l'algorisme procedural utilitzat la GPU, buscant un equilibri entre funcionalitats del programa i eficiència.
- Estudiar possibles modificacions per augmentar els nivells d'aleatorietat de l'algorisme, valorar els canvis que comporten i al seva viabilitat.
- Arrodoniment i finalització de la documentació desenvolupada al llarg del projecte.

1.4 Capítols de la memòria

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

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

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

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

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

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

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

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

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

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

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

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

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

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

– ***Capítol 14. Annex***. Es troben els documents que s'han agut de moure per la seva mida.

2. Estudi de viabilitat

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

2.1 Recursos Tècnics

Aquest treball bàsicament s'ha realitzat sempre sobre la tecnologia d'un mateix ordinador, tenint en compte que el sistema operatiu d'aquest és un Windows 10 de 64 bits i no és compatible ni amb Linux ni amb Mac OS. Pel que fa als components de l'ordinador, és una màquina amb un processador AMD Ryzen 5 de. Pel que fa a la targeta gràfica és un model Nvidia GTX 1080. Disposa de dues memòries RAM DDR4 de 4GB a cadascuna.

2.2 Recursos Humans

Com tot projecte de recerca fa falta una persona que s'encarregui de la recerca i la proposició de nous algorismes per poder assolir l'objectiu. Usualment aquest perfil és propi dels grups de recerca de les universitats.

Així que com a recurs humà únicament és necessària un programador i un investigador, que es pot donar el cas que sigui la mateixa persona.

2.3 Requisits Tecnològics

Aquest projecte es basa amb la utilització de targes gràfiques per fer càlculs i per tant, un requisit indispensable és disposar d'una tarja gràfica dedicada en els ordinadors dels usuaris. A més a més, no serveix qualsevol tarja gràfica, és necessari que pugui utilitzar "Compute Shaders" que són propis de les versions d'OpenGl iguals o superiors a la 4.3. En altre paraules, qualsevol tarja gràfica no integrada més que s'hagi dissenyat després del 2012 compleix tots els requisits.

Ara bé, com més potent sigui el hardware, més capacitat de processament tindrà i per tant, més instàncies o generacions procedurals es podran dur a terme de forma simultània. Així que de manera personal recomano tenir almenys els següents requeriments:

- Tarja gràfica amb 1GB de memòria.
- Memòria ram superior a 1GB.

2.4 Cost econòmic

Pel que fa el cost econòmics s'ha de tenir en compte que s'usen ordinadors personals i per tant no hi ha un cost de compra.

També s'ha de tenir en compte la condició en la que treballen sovint els docents i investigadors de les universitats, així que farem els càlculs en base la situació estàndard d'un treballador amb contracte en un grup de recerca:

- Salari mig d'un treballador en un grup de recerca: 800€/mensuals.

Per altre banda tot el programari utilitzat és de codi lliure i per tant no hi ha un cost inherent.

Així doncs el cost econòmica únicament recau sobre el salari del treballador i/o investigador.

2.5 Conclusió de la viabilitat del projecte

En conclusió, tots els requisits es compleixen sense cap problema ja que el requisit més restrictiu és el tecnològic i qualsevol ordinador modern els compleix. Tampoc hi hauria d'haver cap problema en el cost econòmic ja que és un cost baix en comparació a altres projectes informàtics.

3. Metodologia

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

El que s'ha fet ha estat definir un tipus de metodologia que funcionés bé amb les característiques del projecte, tal com es mostra en el diagrama de flux de la *Figura 1 Diagrama de la metodologia*

.

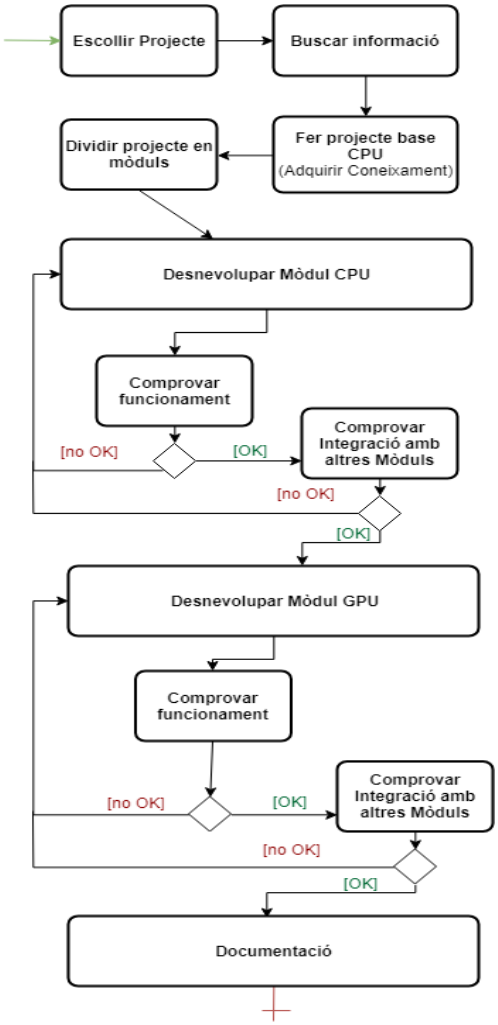


Figura 1 Diagrama de la metodologia seguida

4. Planificació

4.1 Planificació inicial

La planificació d'aquest treball ha estat influenciada per la metodologia de treball escollida. A la primera trobada, Març del 2018, es va decidir que el projecte es faria sobre generació procedural de vegetació inspirats pel videojoc *Horizon Zero Down*, el qual tenia un sistema procedural molt sofisticat però no hi ha la part de generació de vegetació a temps real.

Com que el camp és completament nou i a més s'havien de reforçar conceptes previs, la primera part es va dedicar a l'aprenentatge i investigació dels L-Systems com l'OpenGl, mentre que la segona part es va dividir en dues seccions: la part de CPU i la part de GPU.

Amb tot això dit, les tasques que es van determinar són:

1. Estudiar els L-System.
2. Estudiar OpenGL.
3. Fer un petit projecte base per consolidar coneixements.
4. Desenvolupar el programa de CPU:
    a. Implementar mòdul d'adquisició de dades .
    b. Implementar mòdul L-System.
    c. Implementar Mòdul de rendering.
5. Desenvolupar el programa de GPU:
    a. Adaptar el mòdul d'adquisició de dades.
    b. Implementar el mòdul L-System per GPU.
    c. Adaptar el mòdul de rendering per GPU.
6. Documentació de tot el procediment.

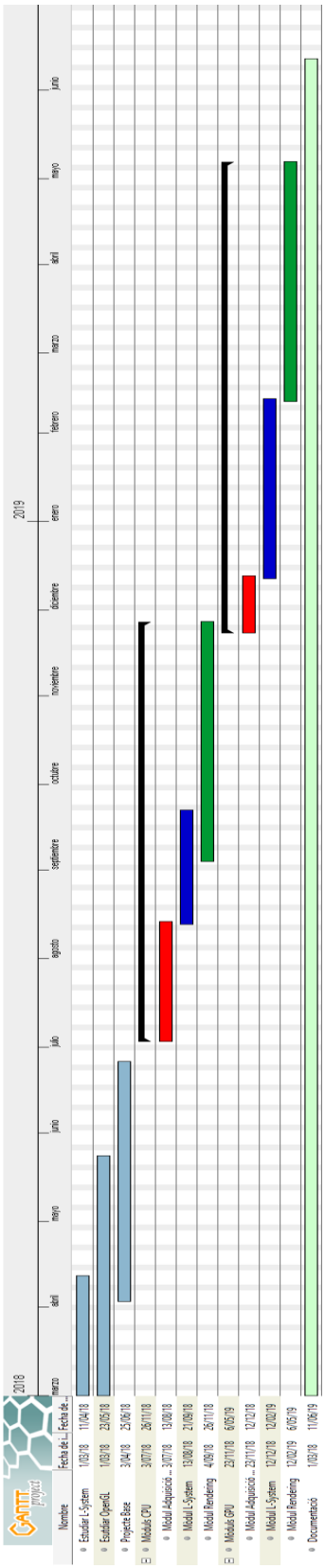


Figura 2 Planificació inicial

4.2 Planificació final

En els projectes és molt difícil aconseguir complir la planificació inicial, ja que sempre hi ha algun contra temps que no s'havia tingut en compte o que simplement es desconeixia que podia aparèixer. Aquests contratemps apareixen sobretot durant la implementació del programa en GPU ja que es treballa sobre terreny desconegut.

Per aconseguir una idea de com es pot implementar una cosa nova, el més important és intentar buscar la lògica que hi ha al darrere i la millor manera és agafar referents d'exemples similars al que es vol aconseguir. Un d'aquests exemples està en jugar a videojocs generats proceduralment i fixar-se com es generen els continguts, per què es generen d'aquesta manera, i en quin moment es generen. També és important provar softwares de vegetació procedural, encara que siguin pensats per CPU, sempre hi ha referències de com està implementat per dins i intentar desxifrar els motius.

Per aquest motiu apareixen més tasques dins dels apartats d'aprenentatge, tant dins dels L-Systems com del OpenGL.

Un altre tasca que no s'havia pensat anteriorment era la comparació entre el model de CPU i GPU. Cal comprovar que els arbres resultants són idèntics i que la gestió dels L-Systems també acaben derivant al mateix resultat.

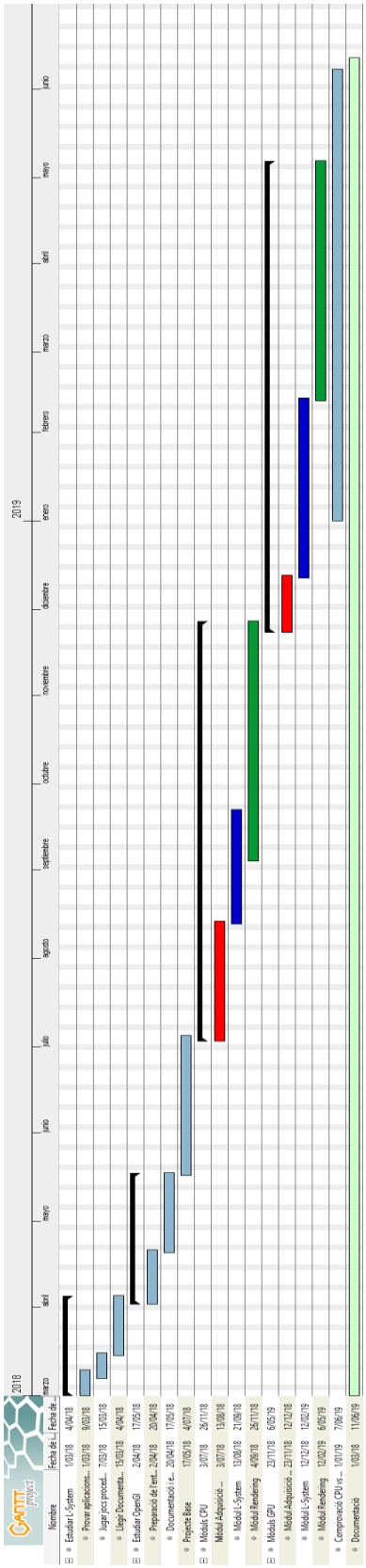


Figura 3 Planificació final

5. Marc de treball i conceptes previs

5.1 GPU

La tarja gràfica és l' element del nostre ordinador que s'encarrega de tota la feina relacionada amb el tractament de gràfics i geometria. Actualment, tots els ordinadors tenen una tarja gràfica, que és la responsable de que podem visualitzar continguts pel monitor del nostre ordinador.

Una GPU és podria dir que és un mini ordinador dins del mateix ordinador, ja que té:

- Una placa base.
- Una memòria RAM.
- Un centre de càlcul o processament.

La diferència principal és en com està dissenyat el centre de càlcul i l'accés a la memòria ram, ja que estan pensats per poder executar un gran nombre de fils en forma paral·lela.

5.1.1 La pipeline gràfica programable

Una de les principals finalitats de la tarja gràfica és el "render" d'imatges per ser visualitzades amb un perifèric. Per aconseguir aquest objectiu s'ha establert una "pipeline" de treball que totes els targes gràfiques segueixen. Aquesta pipeline, pot tenir variacions i ser més o menys complexa, però totes segueixen la següent base:

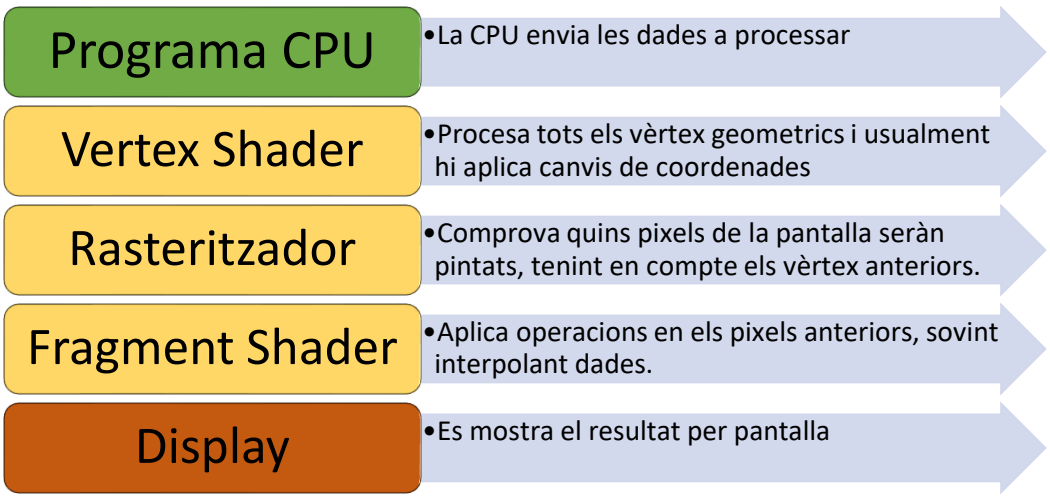


Figura 4 Etapes de la GPU, en aquesta figura es pot veure quins són els passos que fa una tarja gràfica des del moment que rep les dades a processar fins que les mostra per pantalla.

5.1.2 El processador de vèrtexs programable (Vèrtex Shader)

El processador de vèrtexs s'encarrega d'executar els vèrtex shaders o programes de vèrtexs sobre el flux de dades que rep. És la primera fase programable de la pipeline. En aquesta fase únicament és necessari tenir la posició del vèrtex. Per poder començar a fer els algorismes, però sovint també s'aprofita aquesta etapa per realitzar els càlculs de transformació en l'espai entre el món 3D i la càmera, transformacions com rotacions, translacions, escalats entre d'altres. També podem enviar més informació sobre el vèrtex, com ara, el color, la normal, o bé qualsevol tipus de dades que es pugui representar de forma numèrica.

Aquest algorisme s'executa per cada vèrtex en paral·lel, és a dir, que de forma teòrica s'executa el mateix codi per tots el vèrtex de forma simultània. Un cop s'han acabat els càlculs cal enviar informació de sortida, que com a mínim ha de tenir la nova posició del vèrtex, i s'hi pot afegir tots aquells atributs que creiem necessaris.

5.1.3 El processador de fragments programable (Fragment Shader)

El processador de fragments executa els fragment shaders o programes de fragments sobre el flux de dades d'entrada. La principal diferència amb el vèrtex shader és que en aquest cas solem fer càlculs relacionats amb com s'ha de pintar el píxel, i no tant, en quines transformacions geomètriques s'han d'aplicar. Per tant, sovint veiem que en aquesta etapa és el punt on s'apliquen textures i/o càlculs de la il·luminació, entre d'altres.

Un punt important a saber és, que per defecte, el fragment shader fa una interpolació lineal dels atributs dels vèrtex més propers- Per exemple, si el píxel es troba entre dos vèrtex, aleshores el seu color serà la mitjana del color dels vèrtex.

Un cop s'han realitzat tots els càlculs, s'envia la informació a l'etapa final del procés on es descarten tots aquells píxels que no són visibles. És a dir, que part dels càlculs realitzats es descarten.

Tot i això, de la mateixa manera que el vèrtex shader, el fragment també executa el mateix codi per tots els píxels, i en paral·lel.

5.1.4 Compute Shaders

Els "Compute Shader" no estan directament relacionats amb el càlcul de gràfics, sinó que responen a una altre necessitat: utilitzar la potència de càlcul de la GPU per fer càlculs. Així doncs, aquí tenim una llibertat molt gran a l'hora de poder fer els nostres algorismes, on podem tenir les dades d'entrada que volem i treure'n un resultat que pot ser utilitzat un altre cop per la CPU.

Amb altres paraules, els Compute Shader no estan integrats dins la pipeline gràfica que hem vist anteriorment, sinó que respon a la *Figura 5*:

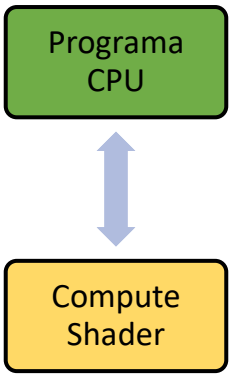


Figura 5 Passos del Compute Shader. En aquesta figura podem veure quins són els passos que fa la tarja gràfica des de que rep les dades fins que processa el Compute Shader.

Aquest funcionament que trenca amb el funcionament normal de la tarja gràfica permet establir un canal de comunicació amb la tarja gràfica i poder entrar i treure'n dades segons ens convingui. Però aquest funcionament ens presenta el següent problema:

Com s'ha dit inicialment, el funcionament habitual de totes les targes gràfiques és la pipeline gràfica. Quan rep una ordre d'executar un compute Shader, ha de trencar amb la pipeline i executar una sèrie d'operacions que estan fora del seu "funcionament normal", i per tant, necessita executar-ho de tal manera que no perdi informació del procés anterior. Per aquest motiu, la GPU fa un petit "Scheduling" del compute shdaer i l'executa quan creu més convenient, això provoca que l'usuari no sàpiga realment en quin moment s'està executant el seu algorisme.

5.1.5 Com funciona una GPU

Fins ara hem vist com podem programar la GPU per tal que executi els nostres programes, i també hem vist que no tots els programes segueixen la pipeline estàndard. El que falta ara és saber com ho fa la nostra tarja gràfica per processar totes aquestes dades.

Per poder entendre el funcionament, primer de tot analitzarem com era l'arquitectura de les targes gràfiques de generacions anteriors, i com s'ha millorat fins a l'actualitat.

5.1.5.a Arquitectura de les GPU fins el 2007

A les primeres targes gràfiques ens trobàvem amb una disposició d'unitats de càlcul especialitzades per a cada tasca, és a dir, hi havia unitats per calcular vèrtex, fragments i textures. Per tant, era una arquitectura pensada especialment per executar la pipeline gràfica.

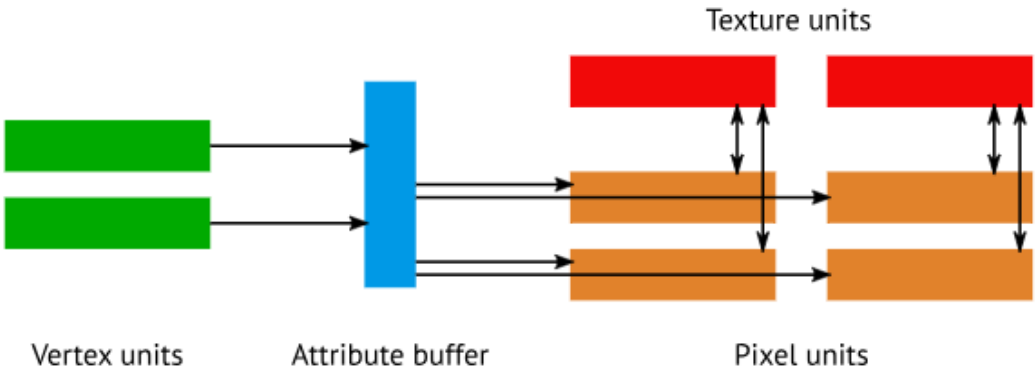


Figura 6 Arquitecutra d'una tarja gràfica fins el 2007

Una característica de doble fil, és que hi havà un nombre limitat de recursos destinats a cada etapa de la pipeline, sobretot es donava molt de pes als fragment o píxel ja que era el pas on s'havia de fer més càlculs i hi havia més elements a analitzar. Per altre banda les unitats dedicades als vèrtex eren molt menors ja que sovint es treballava amb pocs vèrtex.

Així doncs, la tarja gràfica era un sistema molt poc flexible on cada unitat de processament tenia una tasca determinada, i no estava pensada per realitzar altres tasques o funcionalitats.

5.1.5b Arqutiectura post 2007

La següent generació de targes gràfiques van sorgir pel següent motiu: dins del món dels videojocs cada cop hi havia més vèrtex a processar. Els models eren més detallats, els terrenys començaven a estar modelats, i ja no s'utilitzava tant les textures per simular un entorn. Per altre banda, també van començar a aparèixer jocs que utilitzaven al màxim la potència dels fragment o píxels shader, per tant, trobaven que l'arquitectura anterior quedava curta.

La solució proposada va ser decidir que ja no hi hauria unitats de càlcul especialitzades, sinó que totes les unitats tindrien les mateixes prestacions. I per tant, la tarja gràfica podria suportar totes les aplicacions, ja fossin amb una gran quantitat de vèrtex o programes més costosos a nivell de píxel.

Així doncs es va proposar la següent arquitectura:

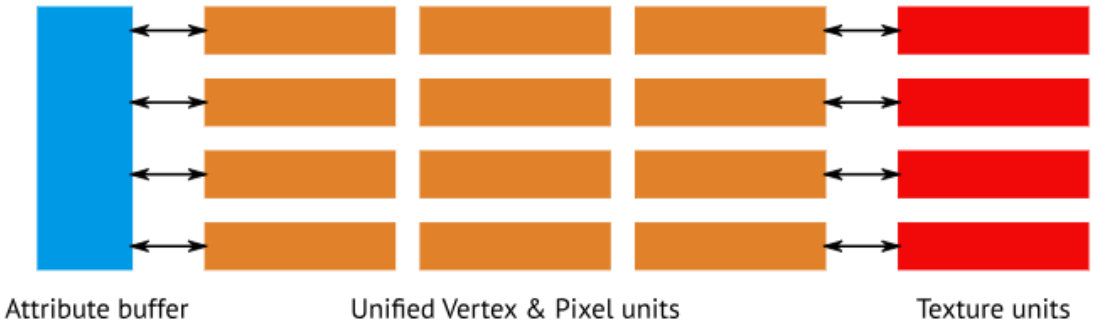


Figura 7 Arquitectura d'una tarja gràfica post 2007

En aquesta arquitectura, tenim unitats de càlcul genèriques i que per tant, podent fer tot tipus de càlcul. Així doncs, aquesta és la primera arquitectura que permetia utilitzar **Compute Shader**. SI ens fixem, "l'Attribute buffer" serveix per guardar els atributs del vèrtex, ja sigui color, normal, etc. I per tant, serveix per guardar les dades locals de cada un.

Per l'altre banda, les unitats de Textura, totes tenen la mateixa textura, el que varia són les coordenades de textura que estan guardades en el buffer d'atributs. Per tant, les unitats de textura són de caràcter global perquè tenen la informació per tots.

Tenint en compte el funcionament esperat de la GPU, ens trobem que els compute shader utilitzen l'arquitectura de la *Figura 8*:

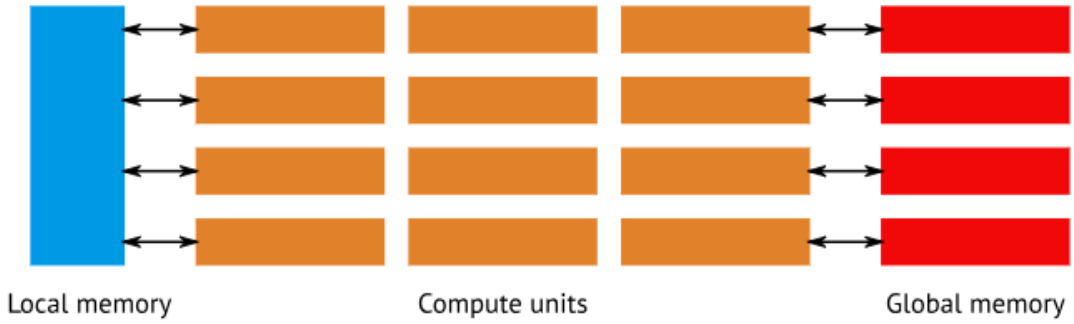


Figura 8 Arquitectura a alt nivell d'un Compute Shader dins de la tarja gràfica.

5.1.5.c Arquitectura per Compute Shader

Quan parlem de Compute Shader necessitem entrar més a fons en l'arquitectura anterior per poder entendre com es distribueix la feina i com s'aprofita al màxim la potència de càlcul.

Primer de tot, donarem més detall de l'arquitectura de la GPU afegint un parell de nivells de memòria Cache i canviant el nom dels Compute Unit per SIMD Units (single instruction, multiple data).

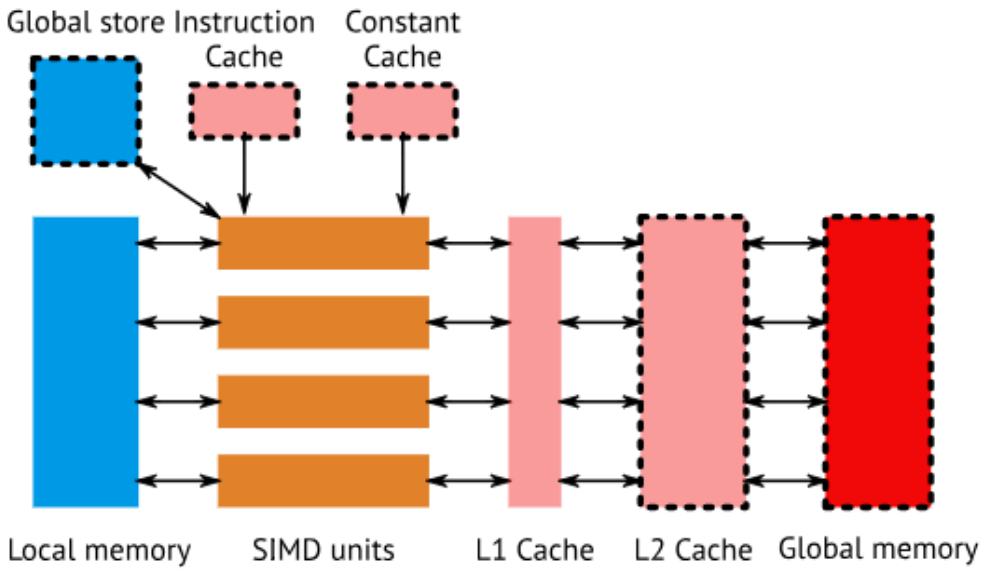


Figura 9 Arquitectura a baix nivell d'un Compute Shader dins de la tarja gràfica.

Recordem que la principal característica de la GPU és el càlcul en paral·lel i per tant, quan fem un algorisme, el volem paral·lelitzar al màxim possible. Malgrat tot, les unitats SIMD no tenen

comunicació entre elles, entre d'altres raons, perquè es podria generar de forma molt fàcil desbordaments de les caches. Aquest fet provoca que no puguem coordinar les unitats SIMD per fer càlcul en paral·lel.

Així doncs, és necessari trobar una manera de poder dividir la feina dins de les mateixes unitats de càlcul i aquí és on apareixen els **grups de treball o "Workgroups"** i **les unitats de treball o "Work Item"**.

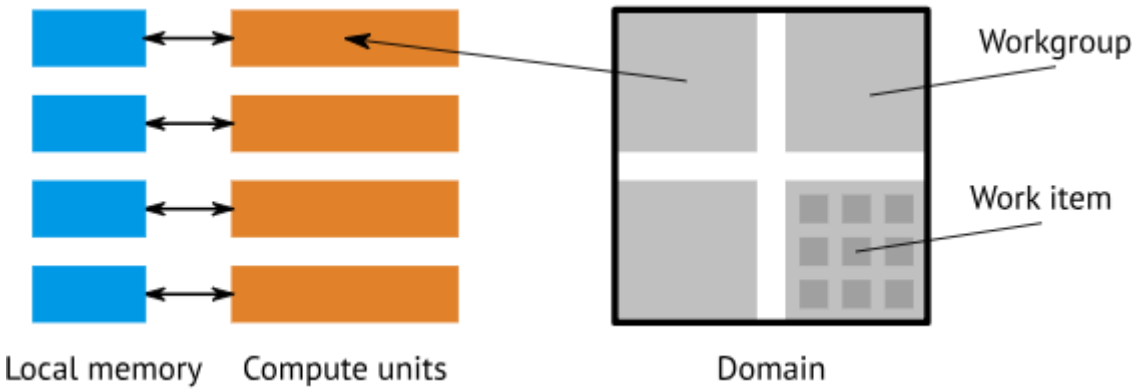


Figura 10 Arquitectura d'un Compute Unit

Domini:

El domini és una feina a fer, és l'algorisme que l'usuari vol executar. Aquesta càrrega la divideix en diferents grups de treball, és a dir, en diferent tasques.

Grup de treball o workgroup

Un grup de treball és un conjunt d'unitats de treball que realitzen una tasca determinada. Cada grup de treball realitza la seva tasca de forma independent.

Unitat de treball o work item

És la unitat que processa la informació, es troba dins d'un grup de treball. Les unitats de treball no segueixen cap ordre d'execució, sinó que s'executen de forma independent.

El següent exemple té la intenció d'aclarir els conceptes anteriors:

"*El nostre algorisme és: donat un conjunt d e matrius, cal aplicar operacions matemàtiques als elements de les matrius*:"

Domini: Volem aplicar unes operacions a totes les matrius.

Grup de treball: Dividirem la feina per matriu, així cada grup de treball s'encarregarà d'una matriu en concret.

Unitat de treball: Dins del grup de treball tindrem tantes unitats com elements té la matriu. Cada unitat realitzarà l'operació corresponent a un element concret de la matriu.

5.2 OpenGL

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

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

5.2.1 OpenGL Shading Language

. En aquesta secció veurem com a partir del llenguatge OpenGL Shading Language (GLSL) podrem programar els anomenats vèrtex i fragment shaders per modificar aquestes unitats. El GLSL, també conegut com GLslang, és una tecnologia part de l'API estàndard d'OpenGL, que permet especificar segments de programes gràfics que seran executats sobre la GPU (shaders). La

contrapartida és el HLSL de DirectX. GLSL és un llenguatge d'ombrejat d'alt nivell basat en el llenguatge de programació C. Originalment introduït com una extensió d'OpenGL 1.4, GLSL es va incloure formalment en el nucli d'OpenGL 2.0 per OpenGL ARB. Va ser la primera gran revisió a OpenGL des de la creació d'OpenGL 1.0 el 1992. Anomenem shader d'OpenGL a un programa escrit en llenguatge GLSL que es pot afegir a la pipeline de processament i permet afegir noves funcionalitats. Hi ha dos tipus de shaders, els vèrtex shaders i els fragment shaders, encara que en general quan parlem de shader ens estem referint a una combinació de tots dos, que mitjançant la realització de còmputs sobre vèrtexs i sobre fragments respectivament ens proporcionaran la manera de programar funcions pròpies per augmentar les capacitats del cicle de producció de gràfics.

5.2.2 Estructura dels shaders

Com s'ha dit anteriorment, el glsl està inspirat amb el C i per tant l'estructura d'un programa és molt semblant a l'estructura d'un programa de C, on apareixen els següents parts:

- Una capçalera indicant la versió d'openGL.
- Una secció amb la declaració de totes les variables d'entrada i sortida.
- Estructures de dades definides per l'usuari, usualment tuples.
- Funcions definides per l'usuari.
- La funció principal o main que és la funció que s'executa al inici.

L'esquema del shader seria el següent:

```
#Versió OpenGL
--Uniforms i variables d'entrada--
--Variables de Sortida—
Definició d'estructures de dades
AltresFuncions(){}
FunciòMain(){}
```

5.2.3 Carregar shaders

Els shaders han de ser llegits des d'un fitxer font contingut en el disc dur, sovint aquests fitxers tenen l'extensió .glsl o bé es pot posar l'extensió .vert i .frag per poder identificar quin és el vèrtex shader i quin el fragment shader.

Siguin quin sigui l'extensió, per poder carregar un shader s'ha de llegir el contingut del fitxer i convertir-lo en una matriu de caràcters, els quals seran llegits i compilats pel compilador tal i com es mostra a la *Figura 11.*

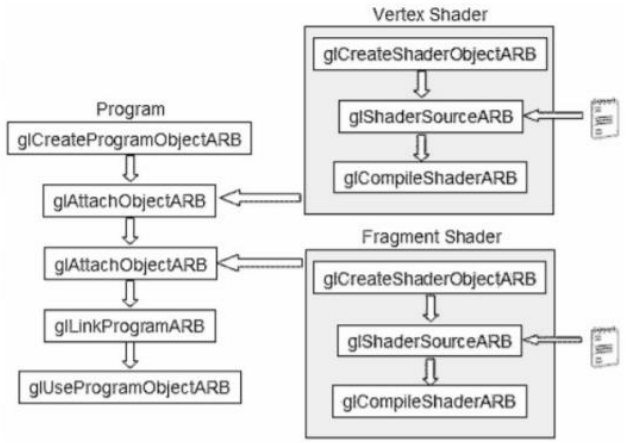


Figura 11 Esquema de la pipeline per compilar un programa complert en OpenGl

5.2.4 Les variables uniforms

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

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

- glUniform{1|2|3|4}{f|i}, per exemple glUniform1i('posició de l'uniform', valor), on aquest valor és un enter (1i).
- glUniform{1|2|3|4}{f|i}v, per exemple glUniform2fv, igual que l'anterior però passem un vector amb 2 floats.
- glUniformMatrix{2|3|4}fv, igual que l'anterior però s'usa per passar mat2, mat3, mat4, o arrays de matrius

5.2.5 Les variables Buffers Entrada i Sortida

Els buffers són idèntics als uniform amb la diferència que les dades es poden llegir i escriure en temps d'execució. A més a més, en els compute shaders, els buffers poden ser d'entrada i sortida permetent que la CPU pugui seguir processant les dades que ha enviat a la tarja gràfica.

La diferència més important apareix a la hora de crear-los i enviar-los a la GPU, els quals es necessiten fer més passos:

1. **Generar el buffer:** amb la comanda glGenBuffers(quantitat, identificador).
2. **Activar el buffer:** amb la comanda glBindBuffer(identificador).
3. **Omplir el buffer:** amb la comanda glBufferData(identificador, mida ,dades);

5.3 Lindenmayer System

El Lindenmayer System, també coneguts com a L-System, és una gramàtica formal i un sistema de re-escriptura en paral·lel. Consta de un conjunt de símbols que tenen associat una o més regles. Cada regla és formada per un conjunt de símbols, de mida "n".

Exemple:

$Simbols$:

$$F\ , X\ , Y$$

$Regles$:

$$F \rightarrow XY$$

$$X \rightarrow XXF$$

$$Y \rightarrow FFY$$

$Original$:

$$FXXF$$

$Generació\ 1$:

$$XY\ XXF\ XXF\ XY$$

La idea principal dels L-System és representar el creixement de les plantes i simular el seu desenvolupament. Malgrat tot, els L-System s'han estès fins a tenir múltiples utilitats, des de la generació de vegetació fins a creació de fractals o en disseny de ciutats.

5.3.1 Algorisme de Tortuga

L'Algorisme de Tortuga té com a referència el llenguatge de "Logo", on l'usuari pot programar el moviment d'una tortuga i veure'n el recorregut que ha fet. En els L-System s'agafa la mateixa idea per passar d'un conjunt de símbols a una geometria 2D o 3D.

És a dir, es comença des d'un punt inicial i aleshores va llegint símbol a símbol, cada un d'ells fa una operació a la tortuga, les més comunes són: **Endavant** , **Girar**, **Guardar Posició** , **Tornar a Posició**. Tots els L-Systems solen representar aquestes operacions amb els mateixos símbols, que solen ser els següents:

Símbol	Operació
F	Endavant (també dit col·locar tronc)
+	Gir positiu en l'eix X
-	Gir negatiu en l'eix X
/	Gir positiu en l'eix Y
\	Gir negatiu en l'eix Y
&	Gir positiu en l'eix Z
^	Gir negatiu en l'eix Z
[	Guardar posició actual
]	Tornar a posició actual

Taula 1 Llistat de simbols usats usualment en els L-System i la seva transformació

Cada operació la identifica una transformació geometria a l'espai que s'utilitza per calcular la posició de la tortuga en tot moment, i quan s'ha de col·locar un tronc, es guarda la posició actual de la tortuga per un futur saber on s'ha de dibuixar.

Un exemple 2D de l'algorisme de la tortuga, agafant les dades de la secció *5.3 Lindenmayer System* i aplicant múltiples generacions, el resultat és el següent:

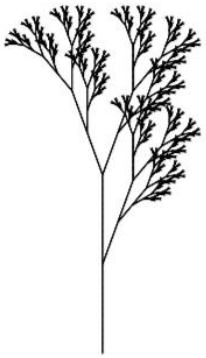


Figura 12 Resultat d'aplicar l'algorisme de tortuga

5.4 Transformacions geomètriques d'objectes 3D

La informàtica gràfica sovint tracta amb objectes 3D que s'han de representar a l'espai, i moure a l'espai. Per aquest motiu hi ha tota una col·lecció d'operacions que són molt recurrents que se'n diuen transformacions geomètriques. Entre elles les més destacades són:

- **Translació**: Moure un objecte en una direcció.
- **Rotació:** Girar els eixos que representen l'objecte.
- **Escalat:** Canviar la mida de l'objecte.

Cada una d'aquestes operacions es defineix amb una matriu 4x4.

5.4.1 Translació

La transformació de translació serveix per desplaçar un objecte 3D per l'espai, aconseguint així un canvi de posició. Aquesta posició sempre es mesura segons el centre de coordenades que de forma estàndard es considera el punt (0,0,0)

La matriu de translació és la següent:

$$\begin{bmatrix} 1 & 0 & 0 & u_0 \\ 0 & 1 & 0 & v_0 \\ 0 & 0 & 1 & w_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

On els valors u, v, w són els valors de moviment en l'eix x,y,z respectivament.

5.4.2 Rotació

La transformació de rotació significa desplaçar un objecte 3D respecte un punt fix. Una particularitat que té aquesta transformació i que la diferencia de la translació, és que tots aquells punts que formen part de l'eix de rotació no veuen la seva posició canviada.

En funció de l'eix de rotació tindrem una matriu de transformació o una altre. En tots els casos la variable és l'angle de gir en radiants.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi & 0 \\ 0 & \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.4.3 Escalat

La transformació d'escalat s'utilitza per canviar la mida d'un objecte ja sigui fent-lo més gran o més petit. Encara que es pugui escalar cada eix per separat, sovint s'aplica el mateix factor d'escalat a tots els eixos per mantenir les proporcions i no deformar el model.

La matriu de l'escalat és la següent:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

On Sx, Sy, Sz són els factors d'escalat per a cada eix x,y,z respectivament.

6. Requisits del sistema?

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

6.1 Plantejament de la problemàtica

La generació procedural de vegetació sempre es fa en offline i mai en temps real ja que té costos de computació massa elevats, això provoca que els escenaris dels videojocs sempre es vegin igual i el jugador perdi l'interès en tornar a explorar una mateixa zona.

Així doncs aconseguir un ambient dinàmic podria ser la solució en les aplicacions on es repeteixen escenaris, ja que l'usuari no tindria la sensació que ja ha estat abans en aquell lloc i sempre veuria diferències.

6.2 Plantejament de la solució

La solució al problema passa per aconseguir fer tot el procés d'un arbre procedural a temps real. Per aconseguir això cal fer ús de la tarja gràfica per processar les dades i aconseguir el màxim paral·lelisma possible.

Així doncs cal agafar tota la tecnologia coneguda dels L-System i adaptar-la

6.3 Requisits funcionals

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

- L'usuari no ha de tenir coneixement sobre L-Systems ni OpenGL, ja que pot utilitzar qualsevol cadena com a Axioma. Ara bé, si vol dissenyar els seus arbres i coneixement de causa, haurà de saber com funcionen els L-Systems.

- Un cop l'usuari sap quins símbols, regles i axiomes vol utilitzar n'hi haurà prou amb executar l'aplicació per veure'n el resultat.

6.4 Requisits no funcionals

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

Aquestes són les funcionalitats que tindrà l'aplicació:

- El sistema ha de ser robust a errors per part de l'usuari i intentar solucionar per ell sol incongruències en l'entrada de dades per part de l'usuari.
- El temps de processament dels arbres s'ha de fer amb el mínim de temps possible, per intentar tenir un renderitzat amb una bona quantitat de frames per segon.

Aquest treball ha estat provat sota una tarja gràfica compatible amb OpenGl 4.3.

7. Estudis i decisions

En aquesta secció explicarem i justificarem la utilització de diferents llenguatges i entorns de desenvolupament que s'han emprat durant aquests projecte.

7.1 C++

El C++ és un llenguatge orientat a objectes desenvolupat els anys 80 i presentat com a l'evolució del conegut llenguatge C. Una de les característiques que destaquen aquest llenguatge és el "tipatge" de les dades, que se'l coneix com un dels menys flexibles i robusts.

7.1.2 Tipatge de les dades

El C++ té diferents tipus de dades elementals entre elles:

- Numèriques: ja siguin nombres enters (int) o bé de coma flotant (float).
- Caràcters (char): representen un caràcter tot i que internament és un enter.
- Lògic (bool): representen un estat lògic de 1/0 o true/false.

Dins d'aquest tipatge m'agradaria entrar més a fons amb les dades numèriques, ja que jugaran un paper important en el treball.

Com s'ha dit abans es pot diferenciar les dades numèriques enteres o les decimals, també anomenades dades en coma flotant. Ja siguin enters o decimals, hi ha diferents tipus de dades en funció de la grandària del numero (en dígits) que volem emmagatzemar.

Abans d'entrar en matèria, cal dir que el C++ no especifica la mida real de cada tipus de dades i deixa que sigui l'arquitectura de la màquina la que decideixi. Ara bé, si que estableix una relació de mides entre els diferents tipus.

Totes les dades poden ser "**signed**" o **"unsigned"**. Si no s'especifica aleshores per defecte s'entenen com a **signed**. La diferència entre els dos és sobre si l'últim bit s'utilitza per indicar si és negatiu o no. Aquesta diferència té un impacte sobre quin rang de dades es permet guardar.

A continuació hi ha dos imatges, la *Figura 13* representa la mida de les dades en funció de l'arquitectura utilitzada, mentre que *Figura 14* representa el rang de dades que pot agafar cada tipus en funció de la mida.

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
`short`	`short int`	at least **16**	**16**	**16**	**16**	**16**
`short int`						
`signed short`						
`signed short int`						
`unsigned short`	`unsigned short int`					
`unsigned short int`						
`int`	`int`	at least **16**	**16**	**32**	**32**	**32**
`signed`						
`signed int`						
`unsigned`	`unsigned int`					
`unsigned int`						
`long`	`long int`	at least **32**	**32**	**32**	**32**	**64**
`long int`						
`signed long`						
`signed long int`						
`unsigned long`	`unsigned long int`					
`unsigned long int`						
`long long`	`long long int` (C++11)	at least **64**	**64**	**64**	**64**	**64**
`long long int`						
`signed long long`						
`signed long long int`						
`unsigned long long`	`unsigned long long int` (C++11)					
`unsigned long long int`						

Figura 13 Taula de les diferents mides de variables segons el tipus de dades numèriques del C++ i arquitectura del sistema.

integer	16	signed	**± 3.27 · 10^4**	**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed	**± 2.14 · 10^9**	**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
floating point	32	IEEE-754	• min subnormal: **± 1.401,298,4 · 10^{-45}** • min normal: **± 1.175,494,3 · 10^{-38}** • max: **± 3.402,823,4 · 10^{38}**	• min subnormal: **±0x1p-149** • min normal: **±0x1p-126** • max: **±0x1.fffffep+127**
	64	IEEE-754	• min subnormal: **± 4.940,656,458,412 · 10^{-324}** • min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** • max: **± 1.797,693,134,862,315,7 · 10^{308}**	• min subnormal: **±0x1p-1074** • min normal: **±0x1p-1022** • max: **±0x1.fffffffffffffp+1023**

Figura 14 Rang de dades de cada tipus de variable numèrica del c++ segons l'arquitectura del sistema.

7.1.3 Gestió de la memòria

El C++ obliga a l'usuari que faci la seva pròpia gestió de la memòria permeten que en pugui demanar i alliberar segons li convingui. Sovint es diferencia dos tipus de memòria, **estàtica** i **dinàmica**. La primera està representada totes aquelles variables que s'inicialitzen i no es poden alliberar fins que s'acaba el programa. En canvi la segona està estrictament lligada amb els punters, i permet demanar o alliberar memòria.

Els punters són referències a memòria, i és responsabilitat del programador que aquestes referències siguin vàlides. L'avantatge respecte la memòria estàtica és que només cal tenir l'adreça de memòria per accedir a les dades, fet que optimitza molt més els programes.

7.2 OpenGL

L'OpenGL és una API que permet renderitzar gràfics 3D, és una de les llibreries més usades dins el món de les aplicacions gràfiques i que et permet treballar amb la GPU a mig – baix nivell.

Un valor afegit és que totes les targes gràfiques actuals suporten l'OpenGL i sovint tenen crides optimitzades a nivell de processador per tal d'aconseguir el màxim rendiment del hardware.

Per utilitzar l'OpenGL cal saber en quin sistema operatiu i quina arquitectura s'executarà, ja que varia en funció de la plataforma. A més a més, cal inicialitzar diferents punters i generar tot un marc de treball per poder fer les crides a la llibreria. Per facilitar aquestes tasques han aparegut diferents llibreries que alliberen al programador de totes aquestes tasques, entre elles **glad** i **GLFW**.

7.2.1 Glad

Aquesta llibreria open-source Glad serveix per inicialitzar OpenGL en el sistema operatiu corresponent. Per fer-ho primer cal obtenir la llibreria correcte de la pròpia pàgina web https://glad.dav1d.de/. Cal tenir present que també s'ha d'indicar quina versió de OpenGL es vol i si cal alguna funcionalitat especifica.

7.2.2 GLFW

Aquesta llibreria open-source serveix per diferents funcionalitats, una d'elles i la raó per la qual l'he escollida, és generar una finestra i un espai on fer els renderitzats. Altres funcionalitats que té és la gestió de processar els esdeveniments de teclat i ratolí que fa l'usuari mentre està executant l'aplicació.

El workflow més senzill de GLFW és el següent:

1. Inicialitzar GLFW.

```
int glfwInit( void )
```

2. Obrir una finestra;

```
int glfwOpenWindow( int width, int height,
    int redbits, int greenbits, int bluebits,
    int alphabits, int depthbits, int stencilbits,
    int mode )
```

3. Comprovar l'entrada per teclat i/o ratolí.

```
int glfwGetKey( int key )
```

```
A_pressed = glfwGetKey( 'A' );
esc_pressed = glfwGetKey( GLFW_KEY_ESC );
```

4. Tancar la finestra.

```
void glfwCloseWindow( void )
```

5. Tancar GLFW.

```
void glfwTerminate( void )
```

Si ajuntem totes les parts:

```
#include <GL/glfw.h>
#include <stdlib.h>

int main( void )
{
    int running = GL_TRUE;

    // Initialize GLFW
    if( !glfwInit() )
    {
        exit( EXIT_FAILURE );
    }

    // Open an OpenGL window
    if( !glfwOpenWindow( 300,300, 0,0,0,0,0,0, GLFW_WINDOW ) )
    {
        glfwTerminate();
        exit( EXIT_FAILURE );
    }

    // Main loop
    while( running )
    {
        // OpenGL rendering goes here...
        glClear( GL_COLOR_BUFFER_BIT );

        // Swap front and back rendering buffers
        glfwSwapBuffers();

        // Check if ESC key was pressed or window was closed
        running = !glfwGetKey( GLFW_KEY_ESC ) &&
                  glfwGetWindowParam( GLFW_OPENED );
    }

    // Close window and terminate GLFW
    glfwTerminate();

    // Exit program
    exit( EXIT_SUCCESS );
}
```

7.3 Assimp

Assimp és l'acrònim de Open Asset Import Library, és una llibreria de codi lliure que permet importar models 3D i escenes 3D. Alguns formats compatibles són: 3D, FBX , BLEND, OBJ, Collada, entre d'altres.

Sempre que carreguem un model mitjançant la llibreria Assimp, es genera l'estructura de la *Figura 15*

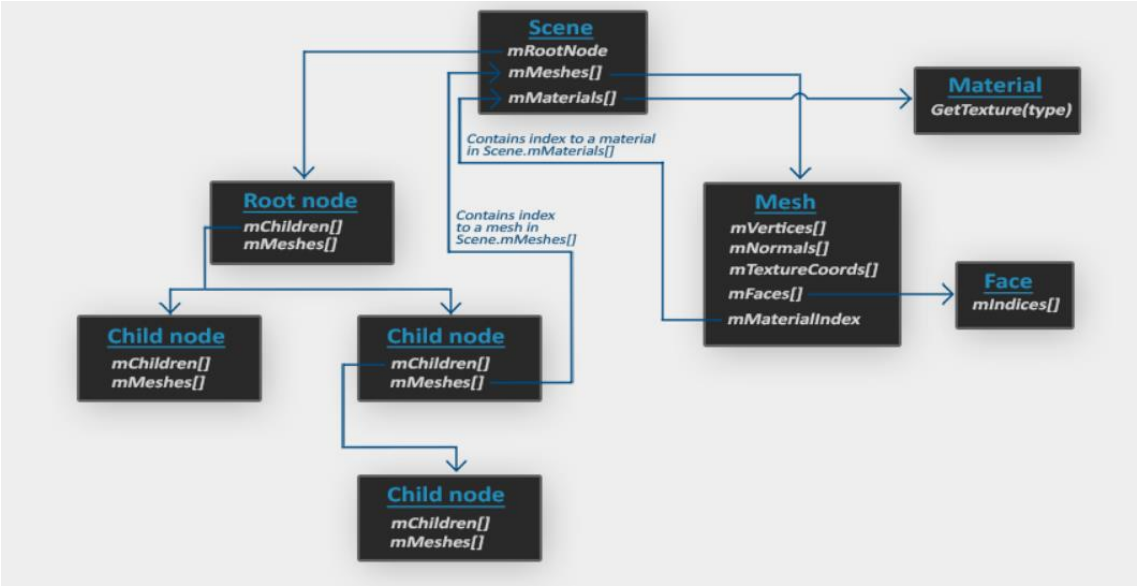


Figura 15 Estructura d'un model carregat amb la llibreria Assimp

Aquesta estructura penja de rootNode que és la base de tots. D'aquí podem accedir a tota la geometria, navegant per l'arbre.

Si només es vol importar un model, com és el nostre cas, aleshores el mateix rootnode te accés a la malla de vèrtex i als materials i/o textures associades.

7.4 Visual Studio

L'entorn de desenvolupament o IDE és el visual Studio, un entorn molt utilitzat ja que permet eines de debug interessants, disposa d'un sistema de auto completació o intellisense que completa les instruccions de manera que no es generen errors de "Spelling". A més a més, permet integrar llibreries de tercers d'una forma fàcil, de manera que permet dur el projecte a altres dispositius sense problema.

També permet incorporar eines de debug per la GPU com el Nvidia Insight que permet veure frame a frame el que passa dins de la tarja gràfica, aquesta eina nosaltres no l'utilitzarem ja que els compute shaders no són compatibles amb aquesta eina.

8. Anàlisi i disseny del sistema

8.1 Introducció

Per la realització d'aquest treball, s'han establert dues etapes: una petit prototip de L-System utilitzant la CPU, i un programa en GPU.

La primera part consisteix en un petit algorisme que calcula un L-System senzill. Aquesta ha servit per fer un primer contacte amb els L-System, aprendre com funcionen, quines limitacions hi ha, i sobretot, intentar preveure quins problemes sortiran a l'hora de passar-ho a la GPU. A més a més, també ha servit per desenvolupar el nucli de visualització d'arbres i poder comprovar-ne el seu funcionament.

La segona part és el volum més important de feina, on es crea tot l'algorisme per renderitzar múltiples arbres en temps real, utilitzant la GPU. A diferència de la part anterior, no hi ha massa informació sobre el tema, i per tant, s'ha desenvolupat l'algorisme i la seva estructura sense poder seguir un referent o un estil concret.

Ambdues parts s'han desenvolupat seguint un sistema modular, és a dir, s'han establert diferents mòduls que es connecten entre ells, i que són totalment independents. D'aquesta manera permet canviar algunes parts sense haver de refer tota la implementació.

Els principals mòduls són:

- Adquisició de dades.
- L-System.
- Sistema de rendering.

La relació entre els mòduls és la següent:

8.2 Diagrama de Classes

8.2.1 Diagrames i fitxes de cas d'ús

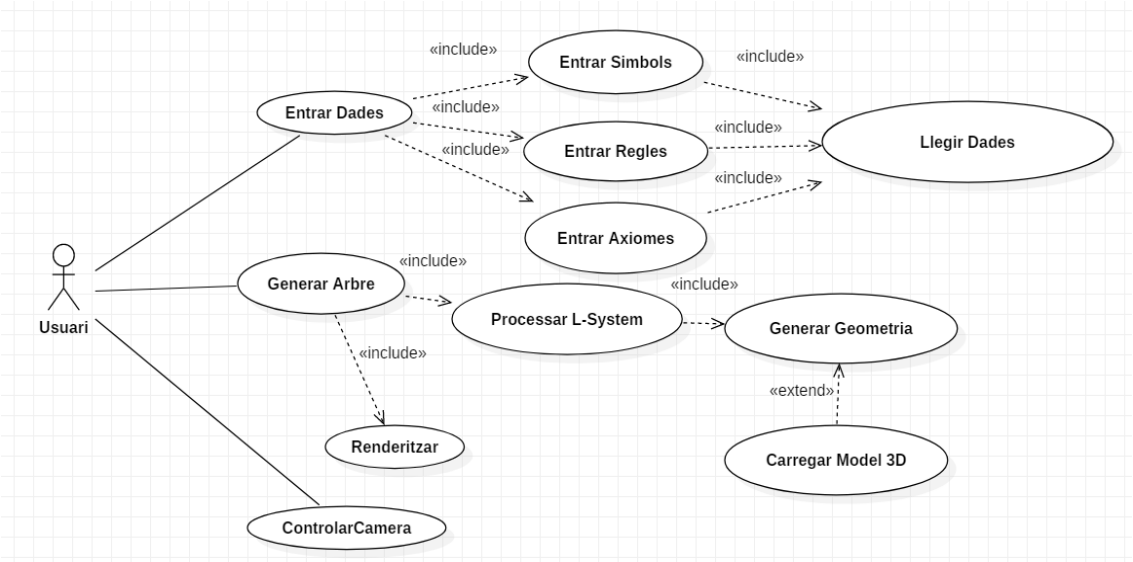


Figura 16 Diagrama de cas dús de l'aplicació

Visió de l'usuari:

La operació d' **Entrar dades** correspon a l'acció de l'usuari quan decideix entrar les dades del L-System en diferents arxius de configuració. Més endavant es donarà més informació de com ha de ser el format dels fitxers.

CAS D'ÚS	Entrar dades
Versió	Visió usuari
Descripció	Un usuari vol entrar les dades de la simul·lació
Actors	Usuari
Precondició	-
Flux principal	1. Obrir el fitxer "Simbols.txt". 2. Omplir amb les dades desitjades. 3. Obrir el fitxer "Regles.txt". 4. Omplir amb les dades desitjades. 5. Obrir el fitxer "Axiomes.txt" . 6. Omplir amb les dades desitjades.
Flux alternatiu	-
Postcondició	Si els formats dels fitxers són correctes, les dades estaran entrades dins del sistema

L'operació de **Generar Arbre,** des de la visió del usuari, està formada per tots els passos que ha de seguir per aconseguir visualitzar l'arbre associat al L-System.

CAS D'ÚS	Generar Arbre
Versió	Visió usuari
Descripció	Un usuari vol generar un arbre.
Actors	Usuari
Precondició	Prèviament l'usuari ha entrat les dades necessàries.
Flux principal	1. Obrir l'aplicatiu. 2. Clicar a Generar Arbre CPU. 3. Seleccionar quantes generacions es volen fer.
Flux alternatiu	-
Postcondició	Arbre generat i mostrat per pantalla.

L'operació **Controlar Càmera** respon a les accions que ha de fer l'usuari per poder moure la càmera mentre està visualitzant un arbre.

CAS D'ÚS	Controlar Càmera
Versió	Visió usuari
Descripció	Un usuari vol controlar la càmera del render
Actors	Usuari
Precondició	Arbre generat prèviament
Flux principal	1. Utilitzar els comandaments (w a s d q e) per entrar les accions desitjades.
Flux alternatiu	
Postcondició	La càmera actualitza la seva posició

Visió del Sistema:

Llegir Dades és l'operació que ha de fer el sistema quan s'inicia per poder entrar i interpretar les dades que l'usuari ha introduit mitjançant diferents fitxers de configuració. Més endanvant es donaran més detalls de com és aquesta entrada.

CAS D'ÚS	Llegir Dades
Versió	Visió sistema
Descripció	El sistema vol llegir les dades entrades per l'usuari
Actors	Sistema
Precondició	-
Flux principal	1. Comprovar que els fitxers de text existeixen i són llegibles. 2. Obrir el fitxer "Simbols.txt". 3. Llegir dades mentre no s'arribi a final de fitxer. 4. Obrir el fitxer "Regles.txt". 5. Llegir les dades mentre que no s'arribi a final de fitxer . a. Comprovar que els símbols del fitxer Regles han estat llegits prèviament amb el fitxer Simbols.txt. 6. Obrir el fitxer "Axiomes.txt". a. Llegir la primera entrada fins a salt de línia.
Flux alternatiu	1. Si els fitxers no existeixen o no es poden obrir, aleshores mostrar error. 2. Si la lectura de Simbols.txt apareix un símbol repetit, ignorar. 3. Si la lectura de Regles.txt apareixen símbols desconeguts o regles repetides, ignorar tota la regla.
Postcondició	Les dades de l'usuari són llegides i carregades correctament.

CAS D'ÚS	Processar L-System
Versió	Visió sistema
Descripció	El sistema rep l'ordre de processar un axioma per generar el L-System corresponent
Actors	Sistema
Precondició	Dades entrades correctament
Flux principal	1. Fer per cada generació: a. Llegir Generació anterior o Axioma, i per cada Símbol fer: i. Comprovar que el Símbol existeix. ii. Buscar si té alguna regla associada. iii. Substituir el símbol per la regla associada. 2. Guardar la cadena final.
Flux alternatiu	-
Postcondició	L'axioma inicial s'ha processat i s'ha convertit en una cadena de caràcters fruit d'aplicar el L-System.

CAS D'ÚS	Generar Geometria
Versió	Visió sistema
Descripció	El sistema rep l'ordre d'associar la geometria al L-System
Actors	Sistema
Precondició	L-System processat prèviament i amb format correcte
Flux principal	1. Crear una transformació 4x4 identitat. 2. Per cada Símbol del L-System: a. Comprovar que el Símbol correspon a una operació unitària. b. Aplicar l'operació unitària a la variable matriu. c. Comprovar si el Símbol és una operació de pintat. d. Guardar la transformació actual. 3. Carregar Model 3D.
Flux alternatiu	1. El Model 3D no té un format correcte es mostra error.
Postcondició	S'ha generat una matriu amb totes les transformacions on s'ha de col·locar el model 3D.

CAS D'ÚS	Renderitzar
Versió	Visió sistema
Descripció	El sistema rep l'ordre de renderitzar el L-System
Actors	Sistema
Precondició	Geometria generada prèviament
Flux principal	1. Inicialitzar tot l'entorn de rendering. 2. Compilar els Shaders per defecte. 3. Per cada transformació calculada fer: a. Enviar la geometria del model 3D i la transformació com a uniforms al shader. 4. Pintar per pantalla.
Flux alternatiu	1. Si hi ha error en compilar els shader, es mostra l'error.
Postcondició	Es visualitzen diferents models 3D col·locats en forma d'arbre.

8.2.2 Mòduls i classes

Com s'ha esmentat abans, el programa consta de 3 mòduls: **Adquisició de dades** , **L-System** i **Sistema de Rendering**. Cada un d'aquests mòduls respon a una necessitat del programa i pot funcionar de forma independent. D'aquesta manera les úniques connexions que hi ha entre els mòduls és per la transferència de dades i informació.

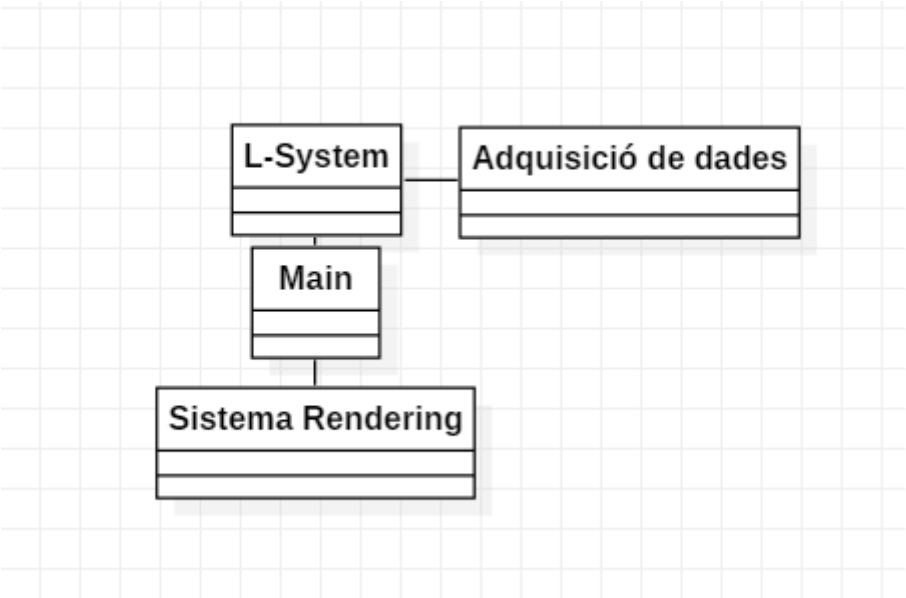


Figura 17 Esquema de mòduls de l'aplicació.

Si entrem dins dels mòduls i observem que no hi ha gaires diferències entre la versió de CPU i GPU a nivell d'estructura del programa. La diferència principal recau amb la classe **TreeRenderer** o bé **ComputeShader** segons si és la versió de CPU o GPU. Aquests dues classes són les encarregades de fer de pont entre el mòdul de L-System cap al mòdul de rendering.

Per altre banda també tenim una classe estàtic de Configuració que únicament guarda els paràmetres de l'aplicació, com per exemple, la llargada dels arbres màxima, l'amplada de les branques màxima, entre d'altres.

A continuació es visualitzen cada un dels diagrames de classe, el de CPU(*Figura 18 Diagrama de classes de l'aplicació per CPU*), el de la GPU (*Figura 19 Diagrama de classes de l'aplicació per GPU*) i la relació entre classes i mòduls(*Figura 20 Relació entre classes i mòdul*):

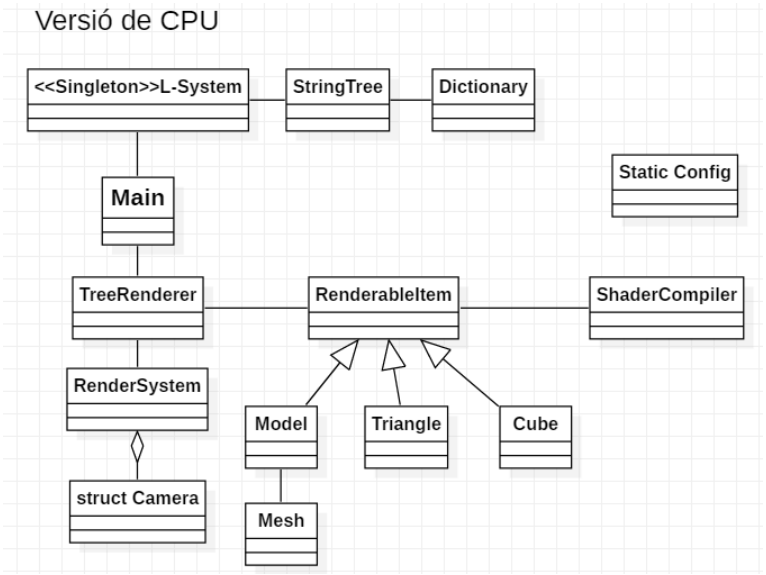


Figura 18 Diagrama de classes de l'aplicació per CPU

En el cas de la GPU podem veure que apareix una classe extra ***LSystemMath*** la qual és necessària pel funcionament del mòdul, però únicament serveix per ajudar amb el processament de dades, i no implica noves funcionalitats. Es podria dir que el conjunt ***LSystemMath*** i ***ComputeShader*** fan el mateix rol que ***TreeRenderer*** en el cas del programa per CPU.

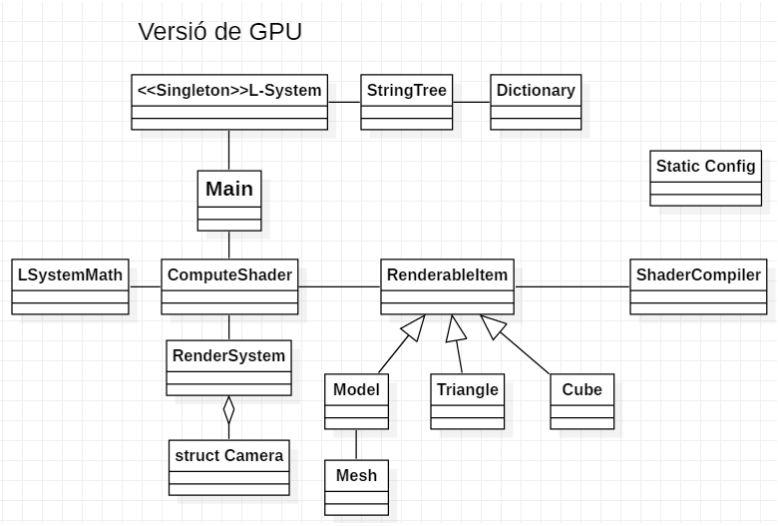


Figura 19 Diagrama de classes de l'aplicació per GPU

Finalment la relació entre classes i mòduls, es pot veure que hi ha una classe sense nom, indicant que tant pot ser ***TreeRenderer*** si estem parlant de CPU o el conjunt ***LSystemMath*** i ***ComputeShader*** en el cas de la GPU. Ambdós casos es troben dins el mateix sistema de rendering i no hi ha diferències funcionals tal i com s'ha esmentat anteriorment.

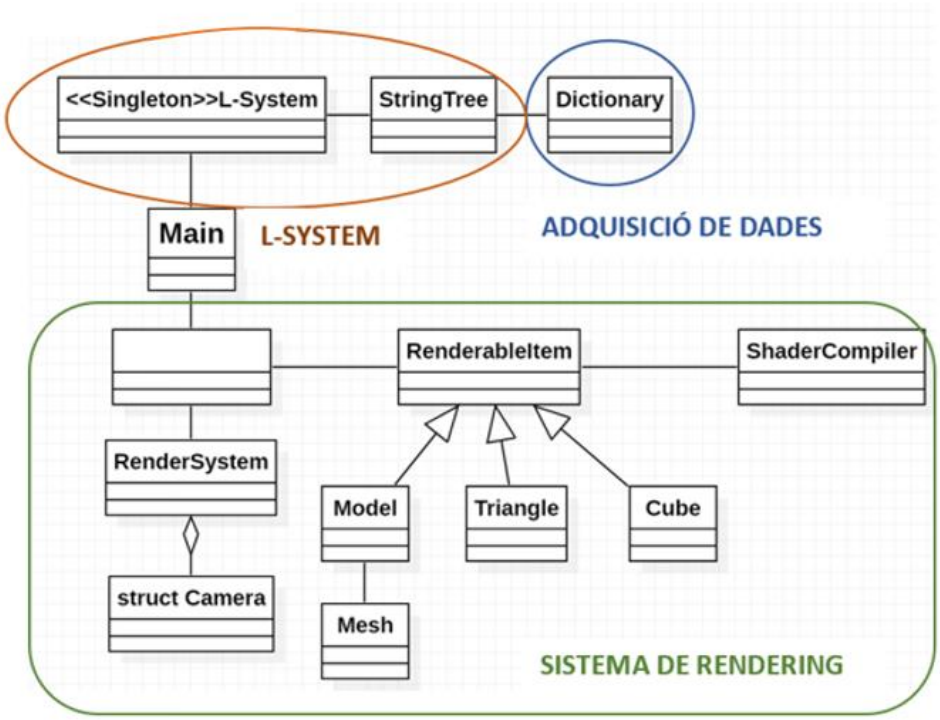


Figura 20 Relació entre classes i mòdul

8.3 Les Classes

8.3.1 Adquisició de dades

El mòdul d'adquisició de dades està estrictament lligat amb el L-System, ja que permet entrar la informació necessària per poder executar-lo. Hi ha diferents dades que depenen directament de l'usuari, entre elles: la gramàtica o símbols del L-System, les regles del L-System i els axiomes o arrels que s'utilitzaran per generar els arbres.

Totes les dades que s'entren seran processades pel programa utilitzant les estructures de dades corresponents. A més a més, es fa un petit control per evitar errors humans a l'hora d'entrar les

dades, però aquest control únicament serveix per evitar errors del programa i no pas per intentar endevinar quina era la intenció de l'usuari.

Les classes que formen el mòdul són les següents:

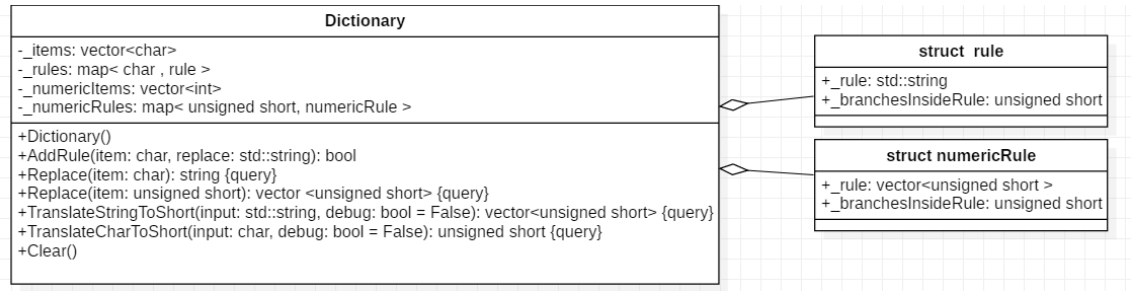


Figura 21 Diagrama de la classe Dictionary

La classe Dictionary és la única que forma el mòdul d'adquisició de dades ja que, com el seu nom indica, serà l'encarregada de traduir els símbols a regles, i per tant , és la classe que necessita saber quins símbols i regles hi ha. La classe està pensada per treballar usant cadenes de caràcters per representar el símbols i les regles, però també s'ha dissenyat per passar de caràcters a números per tal d'optimitzar el procés en cas de ser necessari. En el cas de la CPU es pot treballar amb cadenes de caràcters.

8.3.1.a Atributs i funcions:

Atributs:

- **_items:** és un vector de caràcters que serveix per emmagatzemar els símbols llegits.
- **_rules:** és un map on la clau és un caràcter i el valor una tupla rule, la qual forma part de la mateixa classe, tal i com es pot veure en la *Figura 21 Diagrama de la classe Dictionary*.
- **_numericItems:** és un vector d'enters short per guardar els símbols en format numèric.

- **_numericRules:** és un map on la clau és un enter short, i el valor és una tupla numericRule, la qual forma part de la mateixa classe, tal i com es pot veure en la *Figura 21 Diagrama de la classe Dictionary*

Accions i Funcions:

- **Dictionary():** és el constructor per defecte, el comportament és de configurar la classe usant els fitxers de configuració de l'usuari.
- **Bool AddRule(char item , string rule):** funció que es crida per afegir una regla. Pot ser cridada des de l'aplicació per permetre que l'usuari entri regles un cop ja s'ha iniciat l'aplicació. Retorna si la regla s'ha afegit correctament, o no.
- **String Replace (char item):** donat un caràcter retorna la regla corresponent. En cas de no existir la regla, retorna una cadena buida.
- **Vector<unsigned short> Replace (unsgined short item):** funció que, donat un short, retorna seva regla corresponent en format numèric. Si no existeix, retorna un vector buit.
- **Vector<unsgined short> TranslateStringToShort(string input):** Donada una cadena la tradueix al format numèric.
- **Unsigned Short TransalteCharToShort(char input):** Donat un caràcter tradueix al format numèric.
- **Void Clear():** Neteja la classe i totes les estructures de dades. S'esborren tots els símbols i regles.

A continuació s'explicarà els formats que han de tenir els fitxers de configuració per part de l'usuari per tal de poder entrar les dades dels **Símbols**, **regles** i **Axiomes**.

8.3.1.b Símbols

El fitxer de Símbols serveix per dir al programa quins caràcters s'utilitzaran com a gramàtica per construir el L-System. El programa només processarà aquells caràcters que es trobin dins la llista de Símbols i descartarà aquells que no hi figurin. D'aquesta manera permetem que l'usuari pugui

utilitzar la gramàtica que més li agradi i es trobi més còmode, i al mateix temps, evitem possibles errors fruits de símbols no desitjats que es puguin generar com a resultat d'algun error humà en les regles del L-System.

El format del fitxer serà el següent:

$$NumSímbol\ Caràcter$$

El numero de símbol únicament té la finalitat de servir com a comptador per l'usuari i saber quants símbols hi ha dins el fitxer.

Símbols Base:

Dins el fitxer de símbols, està establert que els 10 primers correspondran a les operacions unitàries pròpies d'un L-System.

Concretament els 10 primers símbols responen a les següents operacions:

Símbol per defecte	**N Símbol**	**Operació**
[	0	*Obrir branca o Apilar transformació actual*
]	1	*Tancar branca o desapilar l'última transformació*
f	2	*Col·locar branca*
F	3	*Col·locar branca*
X	4	*Rotar positivament en l'eix X*
x	5	*Rotar negativament en l'eix X*
Y	6	*Rotar positivament en l'eix Y*
y	7	*Rotar negativament en l'eix Y*
Z	8	*Rotar positivament en l'eix Z*
z	9	*Rotar negativament en l'eix Z*

Taula 2 Simbols per defecte que es carreguen al mòdul d'Adquisició de dades. A la primera columna hi ha el símbol, a la segona el símbol en format numèric, i la tercera l'operació que realitza.

L'usuari pot canviar en tot moment el símbol pel que vulgui, però en cap cas en podrà canviar el comportament.

Símbols Extres:

Un cop entrats els 10 primers símbols, l'usuari en pot entrar tants com necessiti, i aquests no tindran cap comportament o operació associada, sinó que s'utilitzaran com a variables per poder fer les regles corresponents. Una restricció a tenir en compte és que no es poden repetir els símbols, ja que aleshores el símbol repetit seria descartat.

8.3.1.c Regles:

Les regles determinen com s'ha de comportar el L-System a cada generació, i és l'usuari que ha de determinar les seves pròpies regles.

El seu funcionament és simple, sempre que es trobi el caràcter indicat, el substituirà per la regla que hagi declarat l'usuari.

El format del fitxer serà el següent:

$$Símbol : Regla\ (Conjunt\ de\ Símbols)$$

Un exemple:

A : FFF[ZA][FZ]

EL programa no controla que les regles siguin correctes i que no continguin errors gramaticals, és responsabilitat de l'usuari assegurar que la gramàtica és coherent. Tampoc es poden repetir regles que tenen com a base el mateix símbol, ja que aleshores es descartaria.

8.3.1.d Axiomes

Els axiomes són la cadena de Símbols inicials per a cada arbre, i per tant, es podria entendre com la llavor o arrel de l'arbre. Com que a la versió de CPU no ens interessa generar múltiples arbres, aleshores únicament hi pot haver un axioma, si n'hi ha més, es descarten. El format del fitxer serà el següent.

$Axioma\ 1$

$Axioma\ 2$

$Axioma\ n$

8.3.2 L-System

El mòdul L-System és l'encarregat de processar les dades de l'usuari i calcular el L-System corresponent. Com s'ha esmentat abans, l'usuari pot controlar el comportant del L-System determinant les regles i els símbols que s'utilitzaran. El funcionament principal del mòdul és rebre un Axioma d'entrada i treure una cadena de caràcters com a sortida fruit del L-System.

El mòdul està format per dues classes :

- **LSystem (Singleton)**: és la que es comunica amb els altres mòduls i també seria la classe d'accés si altres programadors utilitzessin el mòdul. Simplement fa la funció d'englobar totes les instruccions possibles de forma entenedora i clara.
- **StringTree**: Com el seu nom indica és la representació d'un arbre en forma de string que en el fons és l'essència d'un L-System. És la classe encarregada de generar els L-System i retornar el resultat corresponent.

8.3.2.a LSystem <<Singleton>>

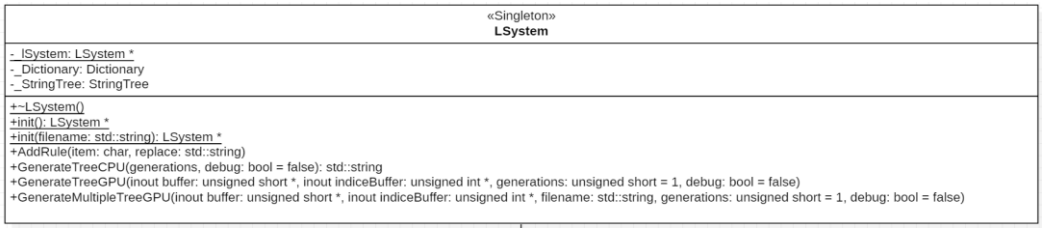


Figura 22 Diagrama de la classe LSystem

Com s'indica, la classe és un singleton i per tant només hi pot haver una instancia creada en tot moment, això és degut a que la classe serveix per processar dades i per tant no té sentit poder tenir diferents instàncies idèntiques que no proporcionen cap avantatges, al contrari, significa una despesa més de memòria.

La classe serveix com a interfície per comunicar-se amb altres mòduls i també com a interfície per a altres programadors que en un futur poguessin utilitzar el mòdul de LSystem.

Atributs i funcions:

Atributs:

- **LSystem *:** punter a la instància de la classe.
- **Dictionary:** Diccionari que farà servir per generar el LSystem de l'usuari.
- **StringTree:** instància de la classe que processa les dades.
- **Axiom:** Axioma que es processarà.

Funcions:

- **LSystem * init():** Funció per inicialitzar la classe i obtenir la referència a la única instància de LSystem.
- **LSystem * init(string filename):** Funció per inicialitzar la classe agafant les dades dels fitxers de configuració de l'usuari. Amb aquestes dades podrà canviar les regles i símbols del seu sistema sempre que vulgui.

- **Void NewAxiom(string axiom):** assigna el primer L-System o Axioma que es processarà. En el nostre cas considerarem que l'axioma és l'arrel de l'arbre.
- **Bool AddRule (char item, string rule):** afegeix una regla nova al diccionari de símbols i regles.
- **String GenerateTreeCPU (int generations, bool debug):** crida per processar les dades i generar el L-System, retorna la cadena de caràcters fruit del sistema de re-escriptura del L-System.
- **Void GenerateTreeGPU(unsigned short * buffer, unsigned int* indices, int generations, bool debug):** Funció per processar les dades i generar el L-System amb un format per ser enviat i processat a la GPU. Utilitza els símbols i regles en format numèric.
- **Void GenerateMultipleTreeGPU(unsigned short * buffer, unsigned int* indices, string filename, int generations, bool debug):** processa tants axiomes com hi hagi al fitxer indicat per filename, i agrupa els resultats per ser tractats a la GPU.

8.3.2.b StringTree

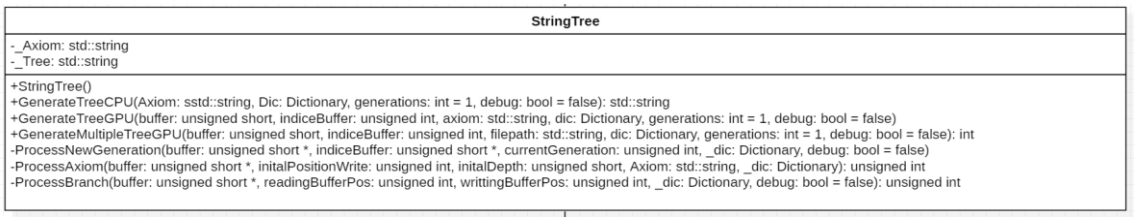


Figura 23 Diagrama de la classe StringTree

La classe StringTree és l'encarregada del processament de les dades, conté tota la lògica necessària per tractar els axiomes i generar els L-Systems corresponents, ja sigui per CPU o GPU.

Atributs i funcions:

Atributs: aquesta classe no té atributs

Funcions:

- **string GenerateTreeCPU(string Axiom, Dictionary Dic, int generations, bool debug):** Funció que calcula un arbre mitjançant les regles del diccionari "Dic". Permet decidir quantes generacions volem fer i ens retorna la cadena de caràcters resultant de l' última.
- **Void GenerateTreeGPU(unsigned short * buffer, unsigned int * indicebuffer, vector<unsigned int> Axiom, Dictionary dic, int generations, bool debug):** Funció que calcula un arbre seguint les regles numèriques el diccionari Dic. Escriu el resultat dins el buffer de totes les generacions, i guarda els índex de cada generació al buffer d'índexs. El format dels buffers segueixen un format eficient per la GPU.
- **Void GenerateMultipleTreeGPU(unsigned short * buffer, unsigned int * indicebuffer, string filename, Dictionary dic, int generations, bool debug):**Llegeix diferents Axiomes d'un fitxer i els processa seguint les regles numèriques del diccionari. Al buffer es guarda únicament l'última generació de cada Axioma, i el buffer d'índexs guarden la posició de cada arbre.
- **unsigned int ProcessNewGeneration(unsigned short* buffer, unsigned short * indiceBuffer,unsigned int currentGeneration, Dictionary _dic, bool debug = false):** Funció auxiliar per a la generació d'arbres en GPU. Es crida cada cop que s'ha de generar un nova generació. La finalitat és agafar la generació anterior, i aplicar les regles corresponents. Finalment, aplicar el format al buffer de sortida.
- **bool GenerateNewBranch(unsigned short * buffer, unsigned short * branches, unsigned int *currentWritablePos, unsigned short *currentDepth, unsigned int initalPositionWrite):** Funció auxiliar per a la generació d'arbres en GPU. Comprova que

es pot generar una nova branca, i que hi ha l'espai corresponent dins el buffer per poder guardar les dades.

- **unsigned int ProcessAxiom(unsigned short * buffer, unsigned int initalPositionWrite, unsigned short initalDepth, string Axiom, Dictionary _dic):** Funció auxiliar per la generació d'arbres en GPU, processa l'axioma inicial donant-li el format corresponent per poder començar a aplicar el L-System.
- **unsigned int ProcessBranch(unsigned short * buffer, unsigned int readingBufferPos, unsigned int writtingBufferPos, Dictionary dic, bool debug):** Funció auxiliar per la generació d'arbres en GPU, processa una branca de la generació anterior per tal de poder aplicar el L-System i generar una nova generació.

8.3.3 Sistema de Rendering

El mòdul de rendering s'encarrega de tots aquells aspectes relacionats amb la visualització d'elements, en el nostre cas, arbres. Per fer-ho cal que inicialitzi tot l'entorn d'OpenGL necessàri, carregar els models apropiats i gestionar la geometria per enviar-la a la tarja gràfica.

A la *Figura 24* podem veure l'estructura del mòdul:

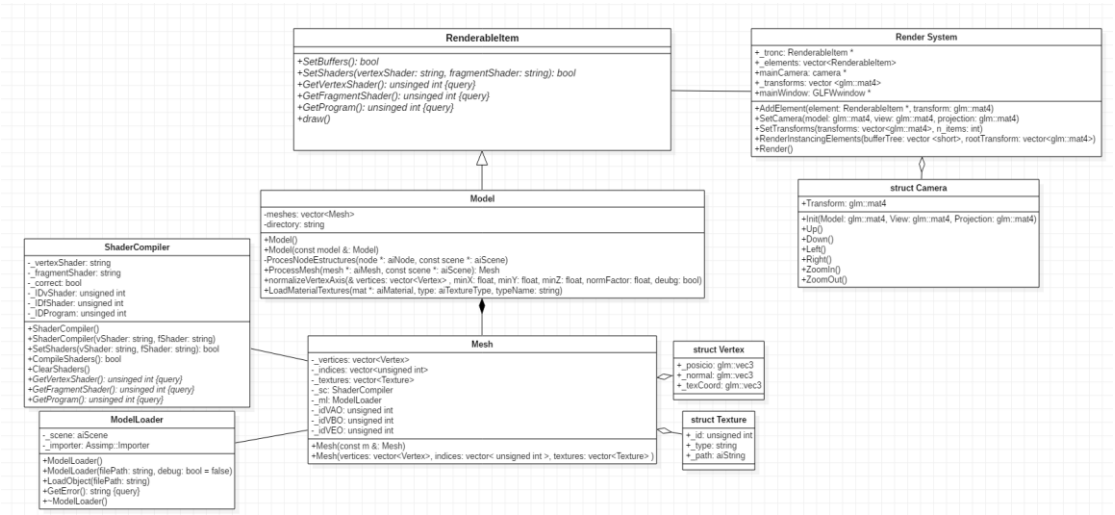


Figura 24 Estructura del mòdul de rendering

8.3.3.a RenderSystem

La classe més important del mòdul és el RenderSystem, ja que és el nucli de tot i també és la que interacciona amb l'exterior. Aquesta classe té el control de la càmera, la finestra on es dibuixa, els objectes que es mostren i permet controlar la càmera per part de l'usuari.

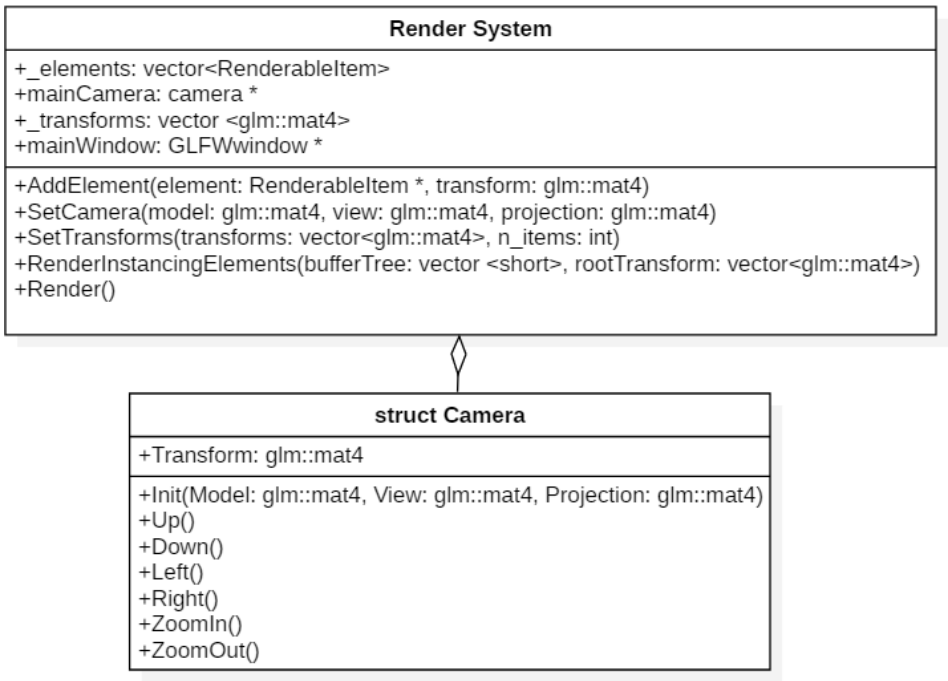


Figura 25 Diagrama de la classe RenderSystem

Atributs i funcions:

Atributs:

- **càmera *:** càmera del sistema de rendering, serà la responsable de controlar el que es visualitza per pantalla.
- **GLFWindow *:** finestra on es dibuixarà el rendering de l'aplicació.
- **Vector<RenderableItem>:** guarda tots els elements que s'han de dibuixar.
- **Vector<glm:**:mat4>: guarda les transformacions dels elements que s'han de dibuixar, on l'índex de l'element correspon amb l'índex del vector de transformacions.

Funcions:

- **RenderSystem():** el constructor per defecte inicialitza totes les variables de l'entorn de rendering, com és la càmera i la finestra on es dibuixarà.
- **SetCamera(glm:mat4 Model, glm:mat4 View, glm:mat4 Projection):** prepara la càmera amb les matrius de model, vista i projecció (sovint conegut com a MVP). Aquestes matrius determinen com serà la càmera i com es dibuixaran els objectes 3D.
- **AddElement(RenderableItem *e, glm::mat4 transform):** Afageix un nou element al vector de renderitzar, cal proporcionar la transformació de l'element per poder dibuixar allà on toca.
- **SetTransforms(vector<glm::mat4> transforms, int n_items):** Funció per poder canviar les transformacions dels objectes. Substitueix les primeres n transformacions al nou vector enviat.
- **Render():** bucle de render bàsic, dibuixa tots els elements entrats utilitzant la funció AddElement(), i permet que l'usuari comenci a controlar la càmera.
- **RenderInstancingElements(vector<short> bt, vector<glm::mat4>rootTransforms):** Aquesta funció està pensada per renderitzar arbres utilitzant la tècnica d'instancació de geometria. Per fer-ho cal que primer s'hagi afegit amb AddElement() tots els troncs a renderitzar. També cal enviar com a paràmetres un vector indicant quants troncs té cada arbre i un vector amb la transformació de l'arrel dels arbres. D'aquesta manera, tots els trossos de tronc, quedaran ben col·locats.

Pel que fa la càmera, podria ser una classe nova però s'ha dissenyat com una struct perquè únicament recull una quantitat d'operacions molt senzilles, en cap d'elles hi ha un processament de dades important, Els seus atributs i funcions són:

Atributs:

- **Glm::mat4 transformació:** guarda la transformació actual de la càmera.

Funcions:

- **Init(glm::mat4 model, glm::mat4 view, glm::mat4 projection):** Inicialitza la càmera amb les transformacions MVP.
- **Up()**: mou la càmera en l'eix Y positiu.
- **Down()**: mou la càmera en l'eix Y negatiu.
- **Left()**: mou la càmera en l'eix X positiu.
- **Right()**: mou la càmera en l'eix X negatiu.
- **ZoomIn()**: mou la càmera en l'eix Z positiu.
- **ZoomOut()**: mou la càmera en l'eix z negatiu.

8.3.3.b RenderableItem

Aquesta classe és la classe abstracta de la qual heretaran totes les classes que representen un objecte a dibuixar per pantalla. Implementa els mètodes basics per poder ser dibuixat i per preparar la geometria per ser enviada a la tarja gràfica.

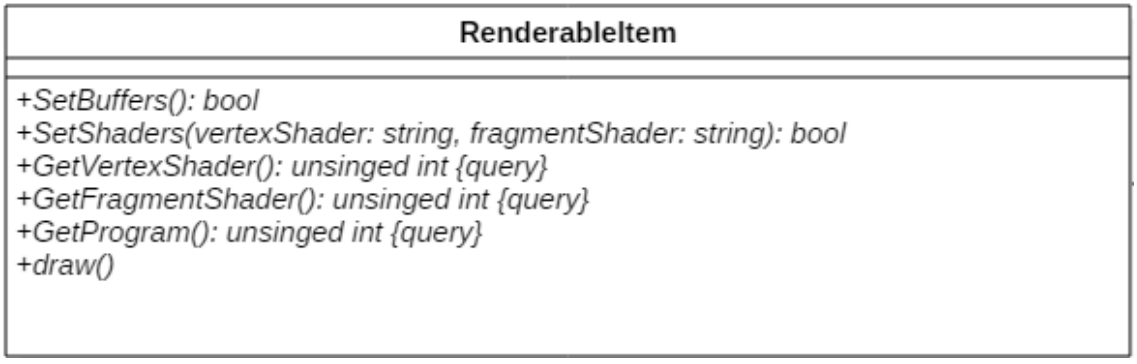


Figura 26 Diagrama de la classe Renderable Item

Atributs i funcions:

Atributs: aquesta classe no té atributs

Funcions:

- **bool SetBuffers()**: Funció que prepara tots els buffers necessaris per enviar a la tarja gràfica, entre ells hi ha: vèrtex buffer, attribute buffer, attribute object buffer.
- **bool SetShaders(string vertexS, string fragmentS):** Funció que, donats els paths a dos programes (shaders), els compila per a la tarja gràfica i prepara per utilitzar-los durant el rendering.
- **Unsigned int GetVertexShader():** Agafa l'identificador que ha donat la tarja gràfica al vèrtex shader un cop l'ha compilat. Si hi ha hagut un error, retorna 0.
- **Unsigned int GetFragmentShader():** Agafa l'identificador que ha donat la tarja gràfica al fragment shader un cop l'ha compilat. Si hi ha hagut un error, retorna 0.

8.3.3.c Model

La classe model implementa tot el necessari per poder mostrar un model 3D carregat d'un fitxer extern. Fa tot el control de les malles de punts, les textures, les normals, i tots els atributs que puguin tenir aquests models 3D.

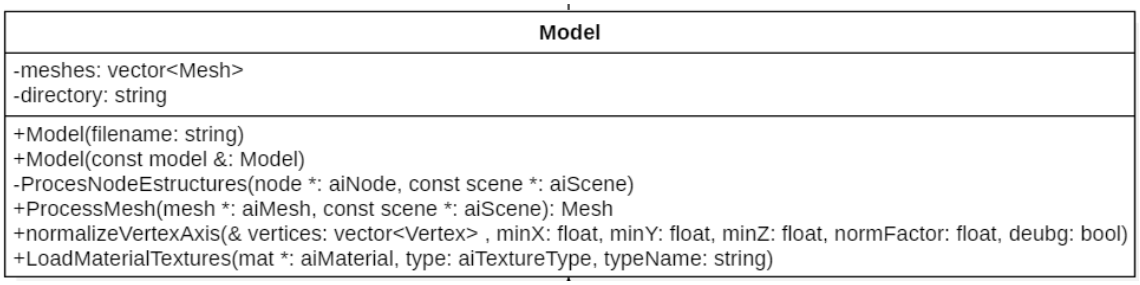


Figura 27 Diagrama de la classe Model

Atributs i funcions:

Atributs:

- **vector<Mesh>:** guarda tota la informació de les malles de punts d'un fitxer. Usualment només és 1.
- **String:** Guarda el fitxer on es carrega el model per si en un futur cal recarregar les dades.

Funcions:

- **Model(string filename):** Constructor base de la classe, obre el fitxer i carrega el model 3D.
- **Model(const Model&):** Constructor còpia.
- **ProcesNodeEstructures(aiNode* , const aiScene *):** Funció que processa l'arbre de nodes que es carreguen quan s'utilitza la llibreria Assimp. Veure *Figura 15*
- **Mesh ProcessMesh(aiMesh*, const aiScene *):** Funció que donada una malla, agafa tota la informació dels punts i construeix un objecte de la classe Mesh.
- **NormalizeVertexAxis(vector<Vertex>&, float,float,float,bool):** Funció que donat un vector de punts canvia l'eix vertical al indicat i normalitza els punts entre 0 – 1 .
- **LoadMaterialTextures(aiMaterial*, aiTextureType, string):** Funció que carrega totes les textures associades al model.

8.3.3.d Mesh

La classe mesh guarda tota la informació necessària per una malla de punts, en el nostre cas l'hem limitat a tenir els punts dels vèrtex, els índex dels polígons i les textures associades. També s'encarrega de preparar els buffers per enviar a la tarja gràfica a l'hora de renderitzar la malla, així com de compilar els shaders.

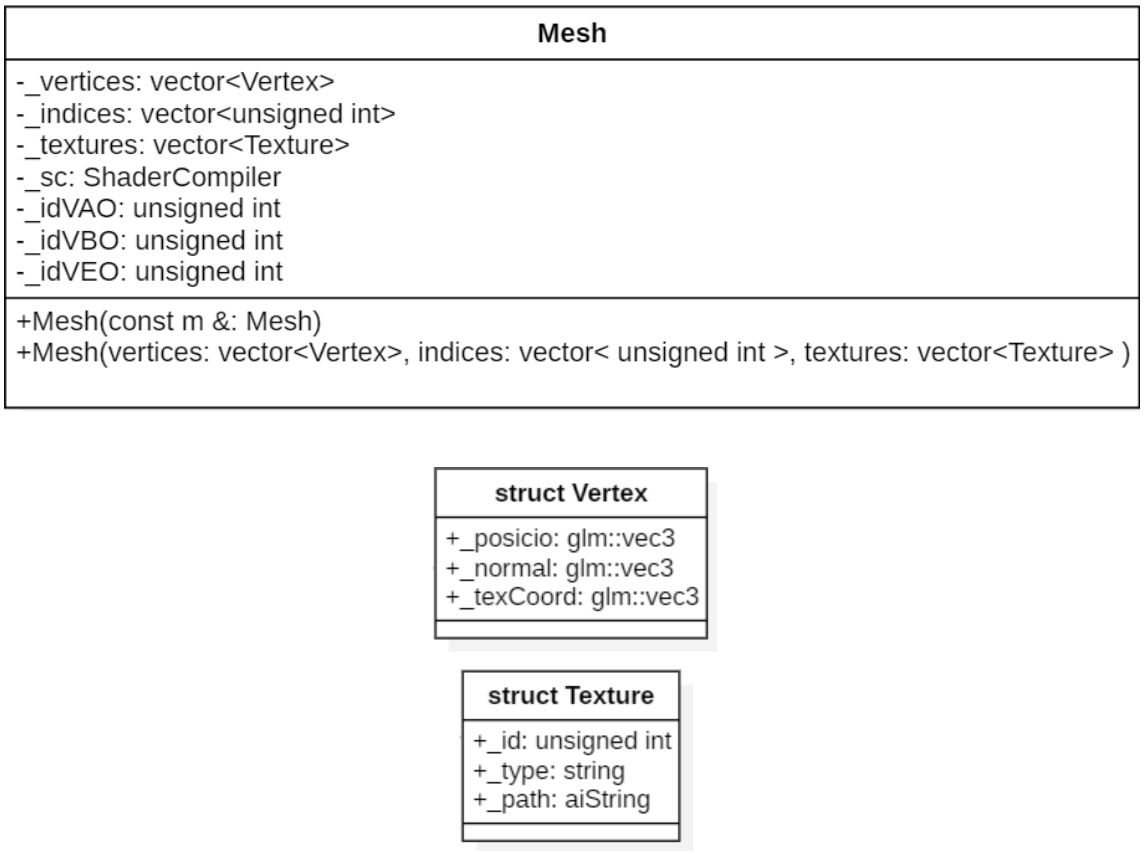


Figura 28 Diagrama de la classe Mesh amb les tuples que conté

Atributs i funcions

Atributs:

- **Vector<Vertex>**: vector que guarda la posició, la normal i les coordenades de textura de cada vèrtex.
- **Vector<unsigned int> :** vector que guarda els índex dels vèrtex dels triangles.
- **Vector<Texture>:** vector que guarda les textures de la malla.
- **ShaderCompiler:** objecte que s'encarrega de la compil·lació dels shaders que es volen aplicar a la malla.
- **Unsigned int:** identificador del buffer (Vertex Array Object), també anomenat VAO.
- **Unsinged int:** identificador del buffer (Vertex Buffer Object) també anomenat VBO.
- **Unsigned int:** identificador del buffer (Element array Object) també anomenat VEO.

Funcions:

- **Mesh(const Mesh&):** Constructor còpia.
- **Mesh(vector<Vertex>, vector<unsigned int> , vector <Texture>):** Constructor amb totes les dades necessàries de la malla.

8.3.3.e Shader Compiler

La classe ShaderCompiler s'encarrega d'agafar els fitxers de text on hi ha el codi dels shaders, llegir-lo i compilar-los dins la GPU.

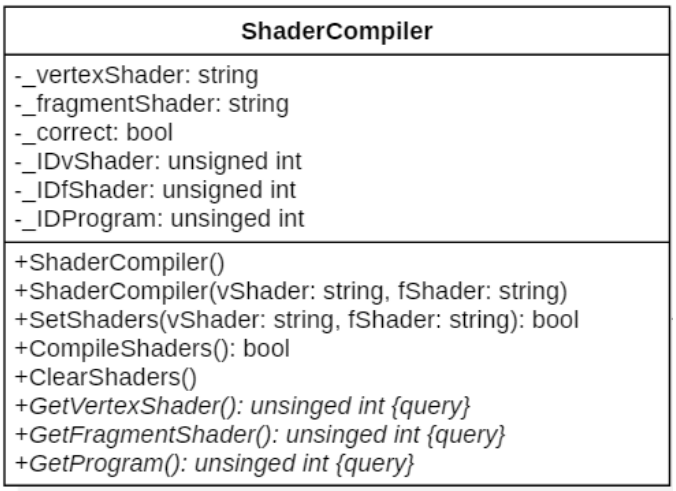


Figura 29 Diagrama de la classe Shader Compiler

Atributs i funcions:

Atributs:

- **String:** emmagatzema el path al vèrtex shader.
- **String:** emmagatzema el path al fragment shader.
- **Bool:** indica si els shaders s'han pogut llegir i compilar correctament.
- **Unsigned int:** identificador del vèrtex shader un cop s'ha compilat a la tarja gràfica.
- **Unsigned int:** identificador del fragment shader un cop s'ha compilat a la tarja gràfica.

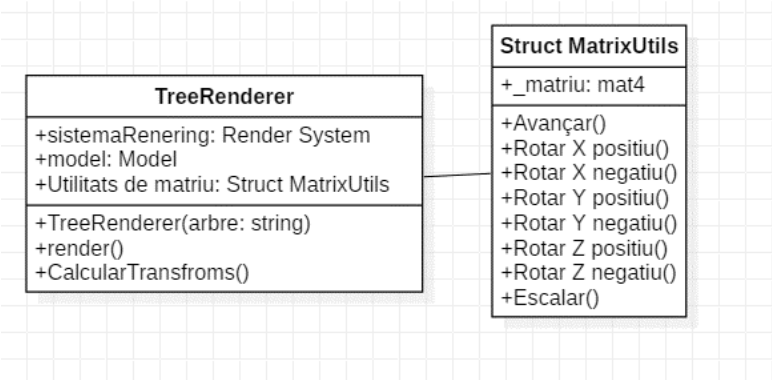
- **Unsigned int:** identificador del programa derivat dels shaders compilats a la tarja gràfica.

Funcions.

- **ShaderCompiler():** Constructor per defecte. Inicialitza totes les variables a valors buits, 0 o fals.
- **ShaderCompiler(string vShader, string fShader):** Constructor que rep el path dels shaders, i inicia tot el procés per compilar-los a la tarja gràfica.
- **Bool SetShaders(string vShader, string fShader):** Actualitza els path dels shaders, elimina la informació anterior així com els compilats anteriors si s'escau, i compila els nous. Retorna si s'han pogut compilar els nous.
- **Bool CompileShaders():** Funció interna que compila els shaders a la tarja gràfica. Retorna si s'han compilat correctament.
- **ClearShaders():**Funció interna que allibera totes les referències als shaders actuals i posa totes les variables a valors buits, zero o fals.
- **Unsigned int GetVertexShader()const:** Funció per obtenir l'identificador del vèrtex shader. Retorna 0 si hi ha hagut un error.
- **Unsigned int GetFragmentShader()const:** Funció per obtenir l'identificador del fragment shader. Retorna 0 si hi ha hagut un error.
- **Unsigned int GetProgram()const:** Funció per obtenir l'identificador del programa. Retorna 0 si hi ha hagut un error.

8.3.3.f TreeRenderer

La classe TreeRenderer serveixp per fer de pont entre el mòdul de rendering i l'exterior. La seva principal funció és rebre les dades provinents del L-System i adaptar-les per poder-les renderitzar. Aquesta adaptació consisteix en passar d'una cadena de caràcters a geometria, per fer-ho utilitza la tuple "Matrix Utils" que li proporciona les operacions bàsiques de transformació espaial.



Atributs i funcions:

Atributs:

- **Render System:** és la referència que té del sistema de rendering i que li permet enviar les dades cap al mòdul.
- **Model:** és el model 3D que carrega per utiltizar com a tronc de l'arbre.
- **Matrix Utils:** tupla auxiliar que li proporciona totes les eines per fer les transformacions necessàries mentre processa el L-System.

Funcions:

- **TreeRenderer(string):** Constructor de la classe on rep l'arbre que ha de processar i adaptar en format de cadena.
- **Render():** Funció per començar a visualitzar els resultats per pantalla. Tot i això comprova que s'ha fet el processament de dades, si no s'ha fet, el fa ell mateix.

- **CalcularTranforms():** Funció que processa el L-System aplicant l'algorisme de la tortuga (veure *5.3.1 Algorisme de Tortuga*), utilitza la tupla MatrixUtils per fer les operacions i guardar el resultat.

Figura 30 Diagrama de classe de Tree Renderer

8.3.3.g ComputeShader

La classe Compute Shader té la mateixa funció que la classe Tree Renderer (veure

8.3.3.f TreeRenderer) però adaptat fer el processament a la GPU, i per poder utiltizar els compute shaders com a eines de processament . Per ajudar a fer el processament té l'ajuda d'una classe associada que li proporciona les matrius de transformació bàsiques dins d'un buffer preparat per enviar a la tarja gràfica.

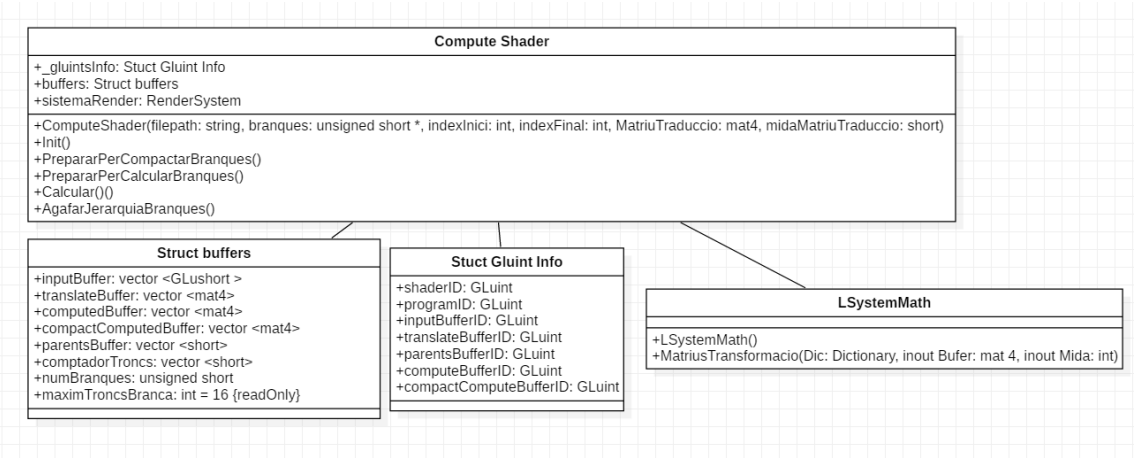


Figura 31 Diagrama de la classe Compute Shader

Atributs i Funcions

Atributs:

- **GluintInfo:** tupla que guarda els índex de shader i buffers de la tarja gràfica.
- **Buffers:** tupla que guarda els buffers que s'utilitzen per enviar a la tarja gràfica.
- **SistemaRener:** és la referència que té del sistema de rendering i que li permet enviar les dades cap al mòdul.

Funcions:

- **ComputeShader(string* , unsigned short*, int, int, mat4,int):** Constructor del compute shader on se li envia tota la informació provinent del LSystem en format de buffes. També comprova que existeixi el path del computeShader.
- **Init()**: Compila el compute shader i processa les dades provinents de l'exterior per adaptar-les als buffers interns.

- **PrepararPerCalcularBranques()**: Prepara tots els buffers necessaris per poder executar el primer computeShader.
- **PrepararPerCompactarBranques():** prepara tots els buffers necessaris per poder executar el segon ComputeShader.
- **Calcular():** executa el computeShader i en processa el resultat.
- **AgafarJerarquiaBranques():** determina l'arbre jeràrquic de els branques que es troben dins el buffer.

8.4 Disseny de l'aplicació

En aquesta secció plantejarem els pseudocodis de l'aplicació per assolir l'objectiu del treball. A més, també s'explicaran més al detall els atributs de les classes. Primer de tot mirarem el programa principal del programa i anirem baixant de nivell fins arribar a les classes de més baix nivell, però sempre mantenint una distància entre la implementació en C++ i pseudocodi, ja que la intenció d'aquest apartat és entendre l'estructura i el funcionament del programa.

Abans de començar, cal tenir en compte que totes les variables que apareixen en aquest pseudocodi segueixen la següent regla:

- Totes les variables que comencen amb _ significa que són atributs de la classe, i que ja han estats especificats i explicats a l'apartat *8.3 Les Classes* .

8.4.1 Main

Primer de tot mostrarem el pseudocodi pel programa pensat per la CPU:

```
Main:
        LSystem * p := inicialitzar( fitxerRegles);
        Axioma:= LlegirAxiomaInicial();
        Arbre:= p->GenerarArbre(Axioma);
        TreeRenderer tr(Arbre);
        Tr.RenderitzarArbre();
```

Com es pot veure, el funcionament és força senzill, cal sempre inicialitzar els mòduls de LSystem i de Render per poder visualitzar l'arbre. S'han intentat fer els mòduls de tal manera que un futur usuari i programador tingui el control sobre el que està fent a cada pas, però que no sigui necessari entrar molt al detall.

8.4.2 Mòdul Adquisició de dades

8.4.2.a Constructor i inicialització del mòdul

El primer de tot cal construir un objecte de la classe Dictionary per poder fer servir el mòdul d'adquisicó de dades. Aquest constructor per defecte llegirà els símbols del fitxer de Símbols:

```
Dictionary::Dictionary:
        Fitxer:= ObrirFitxerSimbols();
        Si FixerNoObert() fer:
                Throw(Error);
        Fi si
        Mentre NoFinalFitxer():
                Num:= LlegirNUmero();
                Simbol:= LlegirSimbol();
                Si SimbolNoRepetit(Simbol) fer:
                        AfegirSimbol(Simbol);
                Fi si
        Fi mentre
```

En aquesta funció llegim els símbols i els números, encara que no se'n faci res d'aquests números. Això és a causa a que el nombre que hi ha al fitxer (que està ordenat de petit a gran), és el nombre que tindrà aquell símbol si el traduïm a dígits. D'aquesta forma l'usuari pot saber com fer la traducció manual, i al mateix temps serveix per poder comprovar errors.

8.4.2.b Inicialització LSystem

Com hem vist anteriorment (veure 8.4.1 Main), és el LSystem el que s'encarrega d'inicialitzar les regles del diccionari, i per tant s'ha inclòs en aquesta secció.

```
LSystem * LSystem::Init(filename):
	Si singletonJaInicialitzat() fer:
		Retorn _lSystem;
	Fi Si
	_lSystem:= LSystem();
	Si _lSystem->llegirFitxer()fer:
		Retorn _lSystem;
	Sinó:
		Retorn nullptr;
	Fi Si
```

Llegir Fitxers de dades

El següent pas és llegir els fitxer de dades que ha entrat l'usuari i omplir les estructures de dades adients. Aquestes dades són les regles que utilitzarà el diccionari.

```
Bool LSystem::llegirFitxer():
	Fitxer:= ObrirFitxer(Regles);
	Si FitxerNoObert() fer:
		Retorn fals;
	Fi Si
	Mentre NoFinalFitxer() fer:
		Simbol:=LlegirSimbol();
		Regla:= LlegirCadena();
		_diccionari.AfegirRegla(Simbol,Regla);
	Fi mentre
```

Afegir una regla

El diccionari és l'encarregat d'avaluar si ha d'afegir o no la regla que li estan proposant. Per decidir si l'afegeix es basa en si existeix el símbol i si aquest ja té una regla associada.

```
bool Dictionary::AfegirRegla(Simbol, Regla):
	Si SimbolNoExisteix() fer:
		Retorn false;
	Fi si
	Si SimbolNoTeRegla()
		AfegirReglaAlDiccionari(Simbol,Regla);
		reglaNum := TraduirRegla(Regla);
		simbolNum := TraduirSimbol(Simbol);
		AfegirReglaDiccionariNumeric(simbolNum,reglaNum);
	Fi si
```

8.4.2.c Intercanviar símbol per regla

Amb les regles carregades, ja podrem utilitzar el L-System amb la gramàtica de re-escriptura, només cal la funció que donat, un símbol ens retorna la regla corresponent:

```
String Dictionary::intercanvi(Simbol):
	_traduccio:= Simbol;
	Si existeixRegla(Simbol):
		_traduccio := regla(Simbol);
	Fi si
	retorn _traduccio;
```

Hem de recordar que no tots els símbols tenen una regla associada, i que també hi pot haver un error humà i introduir símbols no coneguts pel programa. Per tant, s'ha definit que, si no es troba la regla associada al símbol, aleshores es retorna el símbol inicial.

8.4.3 Mòdul LSystem

Aquest mòdul té canvis significatius segons si estem utilitzant la versió de CPU o la versió de GPU. Per aquest motiu i per no ficar massa informació de cop, primer s'exposarà tota la informació relativa al programa en CPU i més endavant s'analitzarà la versió de GPU.

Com s'ha explicat en l'apartat

8.3.2.a LSystem <<Singleton>> la classe L_System únicament és un intermediari entre l'usuari i el mòdul, així que no hi ha una processament de dades inherent. Per aquest motiu veurem directament el funcionament del **stringTree**.

8.4.3.a Generar Arbre

La funció de generar un arbre és la base del mòdul del L-System ja que és la funció encarregada d'aplicar la gramàtica correcte a l'axioma inicial i iterar tantes vegades com faci falta.

```
String StringTree::Generate(Axioma, Diccionari, generacions):
	arbre:= Axioma;
	seguentArbre := "";
	Per cada generació fer:
		Per cada Simbol dins arbre fer:
			seguentArbre += diccionari.intercanvi(símbol);
		Fi per
		arbre = seguentArbre;
		seguentArbre := "";
	Fi Per
	Retorn arbre;
```

8.4.4 Mòdul de Rendering

Abans de començar a renderitzar arbres per pantalla, cal processar les dades provinents del L-System. Principalment, s'ha de fer el pas d'una cadena de caràcters a geometria. Un cop fet aquest procés, aleshores ja es pot començar a renderitzar els arbres, sempre hi quan tinguem l'entorn de rendering inicialitzat.

8.4.4.a Afegir geometria

El procés d'afegir geometria es centra sobretot en llegir l'arbre (cadena de símbols) i aplicar l'algorisme de tortuga propi dels L-System (Veure *5.3.1 Algorisme de Tortuga*). En el nostre cas hem dividit tot aquest procés en dues parts:

1. **Carregar el model base**. Significa carregar tota la informació des d'un fitxer i adquirir-ne les dades importants. A més a més, cal fer un pre-processat per poder aplicar l'algorisme de tortuga sense problemes.
2. **Aplicar l'algorisme de tortuga**. Aquesta algorisme agafarà el model base i el col·locarà tants cops com indiqui el L-System. Per tant, cal que el model es pugui repetir diverses vegades i que no es vegi un inici i un final.

Inicialització i càrrega dels models

La primera part de la feina és molt clara i senzilla d'entendre, només cal carregar el model, compil·lar els shaders que volem associar i carregar els buffers per que la tarja gràfica pugui renderitzar el nostre model. Abans però cal crear un objecte de la classe TreeRenderer:

```
TreeRenderer::TreeRenderer(arbre):
	_arbre = arbre;
	init();
```

```
void TreeRenderer::init():
	_tronc:= new Model();
	_tronc->CarregarModel(nomFitxer);
	_tronc->PreparaShaders(VertexShader, FragmentShader);
	_tronc->SetBuffers();
```

La càrrega en sí del model recau sobre la classe Model, que és l'encarregat de llegir el fitxer corresponent i, amb l'ajuda de la llibreria Assimp, carregar totes les dades correctament.

```
Bool Model::CarregarModel(NomFitxer):
	import:= Assimp::Importer();
	escena:= import->llegirFitxer(NomFitxer);
	Si _escena fer:
		ProcessarNodes(escena ->Node, escena)
	Fi si
```

La funció de processar els nodes utilitza la llibreria Assimp i per tant, cal utilitzar la seva estructura de dades (Veure *Figura 15*).

```
void Model::ProcessarNodes(Node, Escena):
	Per cada malla dins del node fer:
		malla := escena->malles[ node->mallaID ];
		mallaProcessada := ProcessarMalla(malla);
		AfegirMalla( _mallaProcessada );
	Fi per

	Per cada fill del node fer:
		ProcessarNodes( Node->Fill[i] , Escena );
	Fi per
```

Un cop carreguem la malla, cal que la processem tots els vèrtex i extreure'n els atributs que necessitem. En el nostre cas n'obtindrem: la posició, les normals i les coordenades de textura. Recordem que disposem de dues tuples vèrtex i Textura:

- **Vèrtex**: guarda la informació de posició, normal i coordenada de textura.
- **Textura:** guarda la informació sobre l'identificador de textura i la tipologia.

```
Mesh Model::ProcessarMalla (Malla, Escena):
      Vèrtex[]; Index[]; Textures[];
      MaxX,MinX,MaxY,MinY,MaxZ,MinZ;
      Per cada vèrtex dins de la Malla:
            Vèrtex[i].pos := LlegirVertex();
            Si i no és 0:
                  ActualitzarMaxiMinPerCadaEix();
            Altrament:
                  InicialitzarMaxMin ();
            Fi si
            Vèrtex[i].norm := LlegirNormals();
            Si Hi ha textura fer:
                  Vèrtex[i].textC :=  LlegirTextura();
            Fi si
            Per cada cara en malla->Cares fer:
                  Index[i] := LlegirIndexCares();
            Fi per
      Fi per
      X := MaxX – Minx;   Y := MaxX – MinX;   Z := MaxZ – MinZ:
      N := ValorMesGran_X_Y_Z();
      NormalitzarModel(vèrtex[], minX, minY, minZ, N);
      Retorn Mesh (Vèrtex[], Index[], Textures[]):
```

Una altre fase del processament és la normalització de la malla. Totes les posicions dels vèrtex han d'estar entre -0.5 i 0.5, aquesta normalització es fa per tenir el pivot point al 0,0 i per tant, poder aplicar les transformacions sense perill de deformar el model i, a més a més, preparar el model per assegurar que encaixin els uns amb els altres.

```
Void Model::NormalitzarModel(vèrtex[],minX, MinY, MinZ , N):
      Per cada vèrtex fer:
            //Normalitzar entre 0 - 1
            Vèrtex[i].pos.x = ((Vèrtex[i].pos.x – MinX) / N);
            Vèrtex[i].pos.y = ((Vèrtex[i].pos.y – MinY) / N);
            Vèrtex[i].pos.z = ((Vèrtex[i].pos.z – MinZ) / N);
            //Moure el pivot point al centre:
            Vèrtex[i].pos[] -= 0.5;
      Fi per
```

Algorisme de tortuga

El següent pas, amb el model ja carregat, és processar el LSystem que hem generat anteriorment a la funció generarArbre i que s'ha guardat a la variable **_arbre**, i aplicar-li l'algorisme de tortuga. En el nostre cas, anirem calculant la transformació a cada pas i la guardarem cada cop que haguem de dibuixar a pantalla. També tindrem una cua per poder fer les operacions d'encuar i desencuar, on guardarem l'estat de la transformació en certs punts. Cal tenir present que no tots els símbols tenen associats una transformació, veure *Taula 2.*

```
Void TreeRenderer::CalcularTransformacions():
      Per cada símbol dins _arbre fer:
            Si símbol és igual F :
                  _matriu *= Endavant();
                  AfegirTroncRenderitzar(_tronc, _matriu);
                  _matriu*=Esala();
            Altrament si Simbol és igual [ :
                  Encua(_matriu);
            Altrament si Simbol és igual ] :
                  _matriu = Desencua();
            Altrament si Simbol és igual X :
                  _matriu*=RotarXPositiu();
            Altrament si Simbol és igual x :
                  _matriu*=RotarXNegatiu();
```

```
            Altrament si Simbol és igual Y :
                  _matriu*=RotarYPositiu();
            Altrament si Simbol és igual y :
                  _matriu*=RotarYNegatiu();
            Altrament si Simbol és igual Z :
                  _matriu*=RotarZPositiu();
            Altrament si Simbol és igual z :
                  _matriu*=RotarZNegatiu();
            Fi si
      Fi per
```

Sempre que col·loquem un tronc fem un escalat per reduir la mida de la transformació. Gràcies a aquest escalat, aconseguim que l'arbre es vagi fent petit a mesura que creix, de manera que la part baixa de l'arbre és més gruixuda que la part alta.

8.4.5 Render

El render ja és l'últim pas abans no visualitzem el resultat per pantalla. Aquest pas és senzill, però consta de diferents etapes:

1. Inicialitzar l'entorn de OpenGl.
2. Preparar la càmera.
3. Enviar les dades a la tarja gràfica i dibuixar.

8.4.5.a Inicialització de l'entorn de rendering

Per poder utilitzar la llibreria gràfica OpengGL cal inicialitzar primer l'entorn, aquest pas és molt important, ja que no només permet dibuixar a pantalla, sinó que també és l'entorn que permet carregar els models i els shaders a la tarja gràfica. Per aquest motiu, inicialitzarem l'entorn amb el constructor de la classe RenderSystem. Aquesta classe es carrega des de bon principi garantint així que totes les operacions que fem ja tindran l'entorn carregat.

```
RenderSystem::RenderSystem()
	InicialitzarGLFW();
	GladLoadGL();
```

8.4.5.b Càmera:

La càmera serà el que permetrà visualitzar el resultat, i per tant és important tenir una bona configuració. Sinó, podríem no veure res o bé veure deformacions en els models. Recordem que la classe RenderSystem té a dins una tupla anomenada càmera que és l'encarregada de controlar i inicialitzar la càmera.

```
RenderSystem::Camera::init()
	_pos = (3.0 ,3.0 -15.0f);
	ApuntaA(0.0, 3.0 , 0.0);
	_normal = (0.0,1.0,0.0);
	_dreta = calculaDreta();
	Resolució(1280,800);
	_projecció = CalcularP(angle, aspecRatio ,near, far);
	_model = MatriuIdentitat();
	_view = CalcularView();
```

Recordem també que l'usuari pot controlar la càmera mentre s'està renderitzant, i per tant, cal llegir entrades de teclat i actualitzar la càmera.

A continuació trobem el pseudocodi per controlar la càmera des del teclat de l'usuari, s'utilitzen les lletres: W A S D Q E, les quals són pròpies per controlar personatges dins el món dels videojocs, i altres aplicacions 3D.

```
RenderSystem::LlegirEntradaCamera():
       Si TeclaPremuda(W) fer:
              Camera->Amunt();
       Altrament si TeclaPremuda(S) fer:
              Camera->Avall();
        Altrament si TeclaPremuda(A) fer:
              Camera->Esquerra();
        Altrament si TeclaPremuda(D) fer:
              Camera->Dreta();
       Altrament si TeclaPremuda(Q) fer:
              Camera->ZoomIN();
        Altrament si TeclaPremuda(E) fer:
              Camera->ZoomOut();
       Fi si
       Camera->Actualitza()
```

Cada cop que es fa una transformació a la càmera, cal actualitzar un altre cop tots els seus valors per poder veure el resultat per pantalla:

```
RenderSystem::Camera::Actualitza()
       _RecalcularPosicioCamera();
       RecalcularApuntarA();
       ComprovarEixosOrtogonals();
       CalcularView();¡
```

8.4.5.c Uniforms i render Loop

L'últim pas és enviar la informació a la tarja gràfica i renderitzar els elements a la pantalla, en aquesta fase enviarem tota la informació necessària de la càmera i les transformacions calculades en l'algorisme de la tortuga veure *8.4.4.a Afegir geometria*

```
Void RenderSystem::RenderLoop:
       Mentre No TeclaPremuda(Esc):
              LlegirInputCamera();
              Per cada element a renderitzar fer:
                     EnviarInformació(Element, transformació).
                     Element->Dibuixa();
              Fi per
       Fi mentre
```

8.5 Disseny de l'Aplicació amb tarja gràfica:

Durant el projecte es va valorar fer un programa totalment diferent i que fos exclusiu per la tarja gràfica, és a dir, fer totes les fases amb la tarja gràfica i potenciar el càlcul en paral·lel. Malgrat que no hi ha cap referent conegut, es va determinar una forma de fer els càlculs que consistia en generar tot un L-System aprofitant el paral·lelisme de la GPU. Però que es va descartar ja que hi havia problemes en la gestió de la memòria compartida i estructures alternatives del codi.

8.5.1 Proposta LSystem a la tarja gràfica

L'objectiu és poder processar diferents axiomes en paral·lel d'una forma òptima utilitzant la tarja gràfica. La proposta passa per dividir un buffer de la tarja gràfica en blocs de 32 bytes. Cada bloc estarà compost per:

- **Un punter** que apuntarà al bloc "pare" del qual deriva.
- **Espai per 7 unsigned short** que representaran els símbols del LSystem en format numèric.

- **Una marca de nivell de profunditat**. Aquesta marca serveix per dir quin nivell de profunditat de l'arbre pertany aquell bloc. És a dir, si és de nivell 1 (arrel), nivell 2 (primeres branques), nivell 3 (segones branques), etc. Aquesta marca servirà per, després, fer el seguiment de la jerarquia de branques.

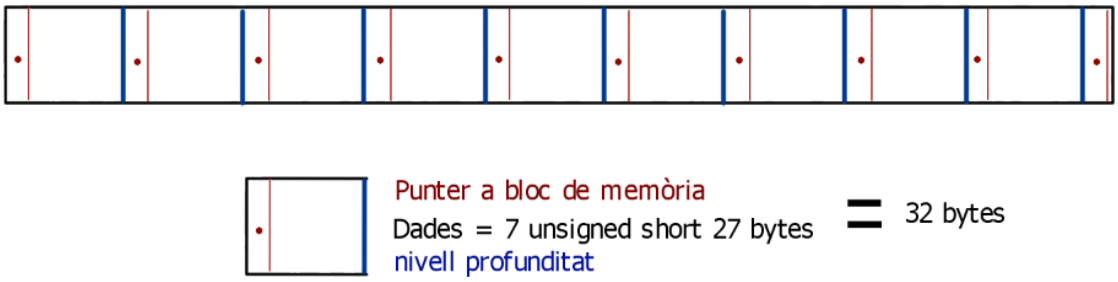


Figura 32 Esquema del format de buffers proposat.

Aleshores el procediment serà el següent:

1. Es Llegirà l'axioma inicial i es posarà al buffer seguint el format de blocs.
2. Es dividirà la feina de traducció de forma que cada nucli tingui un axioma inicial o LSystem.
3. Sempre que s'hagi de fer una branca nova (Operació d'encuar) s'escriurà a un nou bloc de dades, es posarà la referència al bloc anterior i augmentarà en 1 el nivell de profunditat.
4. Sempre que s'hagi de tancar una branca (Operació de desencuar) s'escriurà un nou bloc de dades, es disminuirà en 1 el nivell de profunditat i es buscarà, seguint les referències a nodes, quin és el node més pròxim que té el mateix nivell de profunditat.

Aquest algorisme,al cap i a la fi, fa el processament seqüencial de tot un L-System, amb el problema agreujat que sempre que acaba un nucli s'ha d'esperar que acabin tots abans no s'allibera.

Un altre defecte és que els nuclis de els targes gràfiques no són tant potents que els processadors, i per tant, donada la mateixa feina a un processador és més ràpid i no té el problema anterior.

Amb aquest algorisme no s'acaba de potenciar el paral·lelisme que proporciona la tarja gràfica, la forma d'aconseguir un algorisme potent a nivell de tarja gràfica passa per fer que un mateix L-System sigui tractat per múltiples nucli, obligant al programador a solucionar els problemes de la memòria compartida, la implementació de semàfors òptims, i aconseguir saber on ha d'anar cada branca i aconseguir les referències a cada node de forma correcte.

L'alternativa passa per fer la generació de l'arbre en el processador, aplicant un algorisme molt semblant, fet que ja adaptarà les dades per poder calcular la geometria dins de la mateixa tarja gràfica.

8.5.2 Format de buffer per L-System

Seguint el raonament anterior, la versió de l'algorisme per a CPU té els buffers una mica més canviats, a causa de que el processador té més capacitat de càlcul, els buffers no cal que siguin tant petits. Els buffers tindran les següents consideracions:

- Totes les branques com a molt poden estar formades de 16 símbols.
- Els Nodes de dades, ara tindran una capacitat de 16 unsigned shorts, i per tant, cada Node representarà una branca.
- Cada cop que el tronc es bifurca es consideraran totes com a una nova branca, és a dir, el tronc s'acaba quan troba una branca, aleshores, es genera la branca en qüestió, i una altre branca que seria la "continuació del tronc". Això ens ajudarà a processar les dades dins la tarja gràfica.
- No es permetrà tenir més de 2048 branques per arbre.
- En el model de CPU ja no cal tenir una referència al node anterior.

8.5.3 LSystem:

Un dels primers mòduls que varien respecte el de la CPU és el mòdul de L-System, ja que ara cal processar els LSystem en un buffer amb el format indicat. Així doncs, en comptes de la funció **GenerarArbre** (*8.4.3.a Generar Arbre*), tindrem la funció GenerarMultiplesArbresGPU.

Aquesta funció llegeix els axiomes d'un fitxer, i els processa tots calculant totes les generacions per cada arbre. Un cop calculats, només és queda amb l'ultima generació i la guarda al buffer de sortida. Per altre banda, el buffer de índexs, serveix per anotar a quina posició comença i acaba un altre.

```
int  StringTree::GenerarMultiplesArbresGPU( buffer[], indexs[], fitxer,
        diccionari, generacions):
       posEscriptura = 0; Posicio que comencem a escriure el buffer
       axiomes = llegirFitxer(fitxer);
       Per cada axioma fer:
              bufferAuxiliar[];
       bufferIndexAuxiliar[];
       GenerarArbreABuffer(bufferAuxiliar, bufferIndexAuxiliar,axioma
              Diccionari,generacions);
       //Només ens volem quedar l'última generació
       Final = bufferIndexAuxiliar [generacions -1]
       Si generacions menor 2:
              Inici = 0
       Altrament:
              Inici = bufferIndexAuxiliar [Generacions-2]
       Per i:= inici , inici menor que Final Fer:
              Buffer[posEscriptura] = bufferAuxiliar[i];
              posEscriptura ++;
       Fi per;
       Indexs[ nAxioma ] = posEscriptura;
Fi Per
Retorn numAxiomes;
```

Una funció important les la de generarArbreABuffer, que és l'encarregada del processament de l'arbre en el format esmentat. El funcionament consisteix en llegir l'axioma inicial i adaptar-lo al buffer. Un cop adaptat, cal processar-lo com si fos una generació prèvia a l'actual, i iterar aquest funcionament tantes generacions com necessitem.

```
void StringTree::GenerarArbreABuffer(buffer,index,axioma,diccionari,
       generacions):
       posEscriptura = ProcessarAxioma(buffer,0,1,Axioma,dic);
       mentre correcte i generacióActual < generacions fer:
         correcte = ProcessarGeneracio(buffer,Index,generacioActual,dic);
         GeneracioActual++;
       Fi mentre
```

Primer de tot cal llegir un a un els símbols de l'axioma i comprovar si s'ha d'obrir una branca, tancar-la o bé s'ha de traduir el símbol en format caràcter a símbol numèric i afegir al buffer.

```
int StringTree::processarAxioma(buffer,posEscriptura,Depth,Axioma,dic):
       branques = 0;
       profunditat = depth;
       posInicial = posEscriptura;
       buffer[posEscriptura] = profunditat; posEscriptura++;
       Per tots els símbols dins Axioma fer:
              Si símbol és igual a [: Obrir branca
                     GenerarBranca(buffer, &branques, &posEscriptura,
                                   &Profunditat,posInicial);
              Altrament si Símbol és igual a ]: Tancar Branca
                     Profunditat--;
              Altrament: Escriure la branca
                     Buffer[posEscriptura] = dic.TradueixSimbol(símbol);
                     posEscriptura++;
              Fi si
       Retorn SeguentPuntEscriptura();
```

Fer el processament d'una generació equival a processar totes les branques anteriors i aplicar un altre cop les regles del L-System. Així que la feina en el fons recau sobre el processament de branques.

```
Int StringTree::ProcessarGeneracio(buffer,index,GeneracioActual,dic):
      posEscriptura = Index(GeneracioActual - 1);
      posLectura = Index[GeneracioActual -2);
      num_branques =  (posEscriptura – posLectura) / ampladaMaxima;
      per cada branca fer:
            posEscriptura = ProcessarBranca(buffer,posLectura,
                  PosEscriptura,dic);
            posLectura += ampladaMaxima;
      fi per
      index[GeneracioActual] = posEscripura;
```

La funció processar branca significa processar tots els símbols que formen la branca, tenint en compte que el primer de tots indica la profunditat. Per cada símbol s'haurà d'aplicar la regla corresponent i escriure-la al buffer.

```
int StringTree::ProcessarBranca(buffer,posLec,posEsc,diccionari):
      LecturaOffset = posLec;
      EscripturaOffset = posEsc;
      Profunditat = buffer[LecturaOffset];

      Mentre per cada simbol a la brancafer:
            Regla = dic.Intercanvi(Símbol);
            Si Regla es igual a Obrir branca:
                  GenerarUnaNovaBranca();
                  Profunditat++;
            Altrament si regla es igual a tancar branca:
                  Profunditat--;
            Altrament:
                  EscriureBranca(buffer,profunditat,posEscriptura);
                  PosEscriptura++;
            Fi si
      Fi mentre
      Retorn nBranquesCreades();
```

8.5.3.a Resum del processat dels buffers

Totes les funcions anteriors són per processar el LSystem i adaptar-lo als buffers, per poder aclarir i acabar d'explicar el concepte general, tenim un exemple en la *Figura 33* Podem veure que hi ha tres columnes:

1. La primera representa el format tradicional d'un LSystem processat per CPU.
2. La segona és una representació gràfica del LSystem.
3. És el format que tindria amb els buffers per la GPU.

Símbol	**Regla**
A	AB
B	A [B C]
C	C

Taula 3 Simbols i regles de l'exemple de L-System

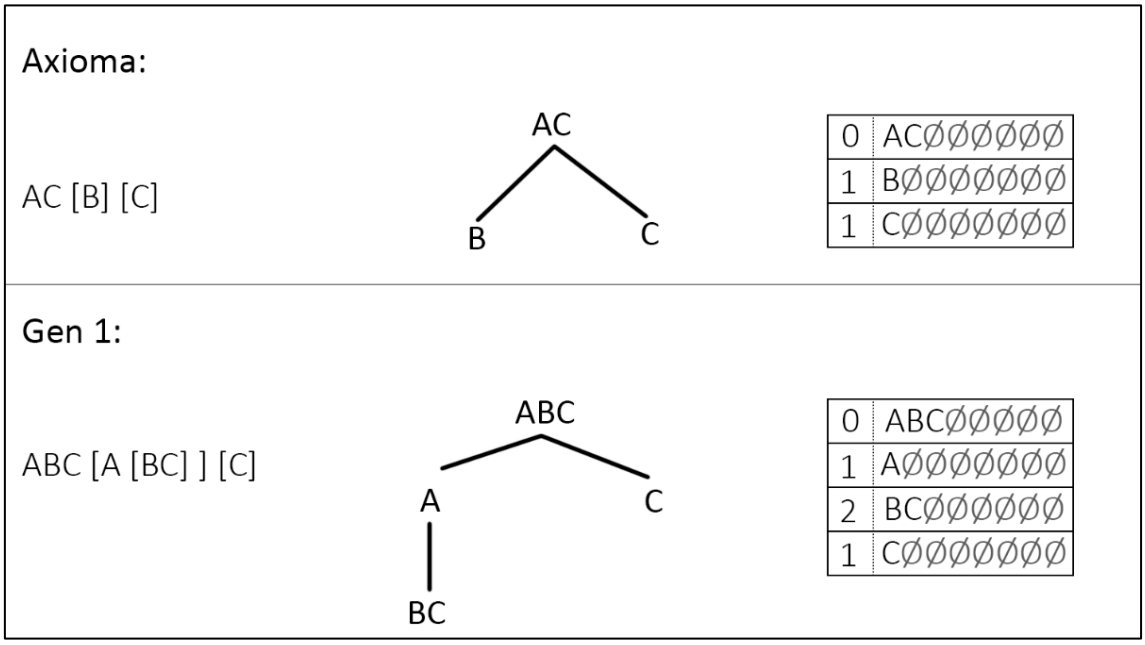


Figura 33 Exemple del processament d'un L-System utilitzant els buffers per la GPU

8.5.4 Mòdul de Rendering

El mòdul de rendering de la GPU té la mateixa finalitat que en la CPU, calcular la geometria necessària i dibuixar-la per pantalla. La diferència es troba en que aquest cas tenim uns buffers

pensats per ser utilitzats en la GPU. És en aquesta etapa que es fa ús dels compute shaders per treure'n el màxim rendiment. En concret s'utilitzaràn dos computes shaders.

1. Calcular les transformacions a nivell de branca.
2. Calcular les transformacions a nivell d'arbre.

8.5.4.a Càlcul de transformacions a nivell de branca

Si recopilem, tenim un buffer on cada bloc de 16 unsigned shorts equivalent a una branca. Aleshores, sabem on comencen i on s'acaben les dades del buffer i per tant, podem fer que cada unitat de càlcul de la GPU calculi una branca determinada, tal i com es veu a la *Figura 34.*

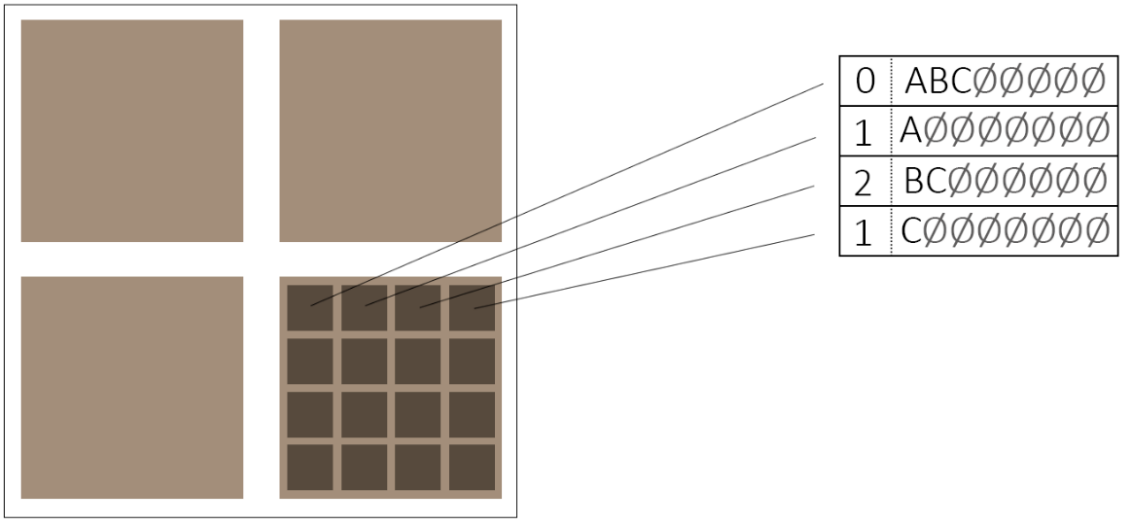


Figura 34 Distribució de la feina dels work items en relació els buffers d'entrada

El compute shader tindrà dues entrades i una sortida:

- **Entrada:** Buffer amb les dades a processar.
- **Entrada**: Buffer de matrius de transformació, tantes com símbols ha entrat l'usuari en el mòdul d'adquisició de dades. On les primeres matrius compleix la transformació que s'especifica a l'apartat *8.3.1.b Símbols*, i la resta són la matriu identitat.
- **Sortida:** Buffer amb les transformacions de cada tronc de la branca. És el resultat d'aplicar l'algorisme de la tortuga, Veure *5.3.1 Algorisme de Tortuga*

El buffer de sortida tindrà el mateix format de buffer que el que hem utilitzat fins ara, però amb petites diferències:

- En comtpes de guardar unsigned shorts, guardarà matrius 4x4.
- La primera posició en comptes de guardar la profunditat, guardarà el nombre de troncs (o matrius valides) que hi ha dins de la branca.
- Les posicions que sobren, (la branca no estava plena de troncs), s'ompliran amb la matriu identitat.
- L'última posició guarda l'última transformació calculada.

El funcionament del compute Shader consta en iterar sobre tots els símbols de la branca que li toca processar i aplicar les transformacions associades. Sempre tenint en compte que només hi poden haver 14 troncs per branca i que la primera i l'ultima posició estan reservades tal i com s'ha explicat anteriorment. Es trobarà més informació a la *secció 9.3.2 Càlcul de transformacions a nivell de branca*.

```
ComputeShader 1:
	MatriuActual := MatriuIdentitat();
	Troncs := 0
	Per Cada element a la branca fer:
		MatriuActual = MatriuActual *bufferTransformacions[element]
		Si element == 3: Afegir tronc
			MatriuActual = MatriuActual * Escalat();
			BufferSortida[ troncs +1]= MatriuActual;
			UltimaMatriu = MatriuActual;
			Troncs++;
		Fi si
	Fi per
Si no hi ha hagut 14 troncs, s'omplen les posicions amb la matriu identitat
	OmplirForatsAmbMatriuIdentitat();
L'ultima posició es col·loca la matriu de l'últim tronc.
	BufferSortida[UltimaPosicio] = UlimaMatriu;
```

En aquest shader cal remarcar la importància de tenir els símbols en format numèric i que els 12 primers símbols equivalen a transformacions geomètriques, d'aquesta manera podem enllaçar de forma directa entre Simbol (numèric) i la transformació que comporta. Si el símbol té un valor superior a 12, aleshores es trobarà amb la matriu identitat, i per tant, no hi haurà cap impacte amb els càlculs.

Un altre punt a tenir en compte és que tot i que tots els shaders escriuen el mateix buffer de sortida de forma en paral·lel, no es solapen ja que cadascú coneix els índex on ha d'escriure. La raó és que el nucli de càlcul, sap quina branca està executant, i per tant, sap quin espai de memòria li pertoca.

8.5.4.b Càlcul de les transformacions a nivell d'arbre

El següent compute shader té dues finalitats: calcular les transformacions finals de cada tronc i netejar el buffer d'espais "en blanc", concretament totes aquelles matrius identitat que hi ha dins les branques, i aconseguir compactar el buffer. Tot i això, abans d'executar el compute shader, hi ha un pas de pre-processament que es fa en CPU, aquest pas busca el pare de cada branca per poder generar l'arbre de forma correcta.

Buscar pares

Aquest pas consisteix en crear un vector de les mateixes dimensions que branques tenim, i a cada posició del vector indicar l'índex de la branca pare que li tocaria aquella posició, és a dir, si la branca 3 penja de la branca 1, aleshores a la posició 3 del vector s'hi col·locarà un 1.

L'algorisme elaborat per aquesta tasca, és una mica complex: es tracta de tenir un vector auxiliar on cada posició indica el nivell de profunditat de l'arbre. Dins d'aquest vector s'hi guarda l'ultima branca que tenia aquest nivell. D'aquesta manera, tindrem guardades les branques més recents a cada nivell de profunditat, i per tant ,quan es genera un fill nou, l'índex d'un nivell superior serà el pare.

```
Void ComputeShader::BuscaPares():
	vectorAuxiliar[]
	Per cada branca fer:
		Profunditat = LlegirProfunditatBufferLSystem();
		vectorAuxiliar[profunditat[ = branca;
		VectoPares[indexBranca] = vectorAuxiliar[Profunditat -1];
	FI per
```

Un cop tenim el vector amb l'índex dels pares de cada branca ja podem executar el nostre compute Shader

Compute Shader

Aquest últim pas abans no tenim les transformacions finals dels tronc, s'explica el motiu pel qual anteriorment en el primer compute shader (Veure *8.5.4.a Càlcul de transformacions a nivell de branca*) ens guardàvem la matriu de l'últim tronc a la ultima posició de cada branca.

Per començar repartirem la feina de la mateixa forma que l'anterior, cada nucli calcularà una branca diferent. Però amb la diferència que abans teníem el buffer quadrat en branques de 16 matrius, en aquest cas volem compactar el buffer i per tant haurem de saber quants troncs hi haurà abans de que podem escriure el buffer, i això es farà mirant la primera matriu de totes les branques anteriors, que indica el nombre de troncs que hi ha a la branca. D'aquesta manera es pot calcular quin espai de memòria pertoca al nucli.

I per acabar necessitem saber la matriu del pare de la branca que estem processant, per fer-ho tenim el vector de pares calculat prèviament que ens indica quina branca hem de mirar. Només caldrà mirar l'última posició.

En aquest cas el nostre compute shader tindrà 2 entrades i 1 sortida:

- **Entrada:** buffer amb les branques provinents del primer computeShader.
- **Entrada:** vector amb els pares de cada branca.
- **Sortida:** buffer comprimit amb les transformacions dels troncs finals.

El seu funcionament consta de tres etapes, la primer cal saber quants troncs hi ha abans dels que li toquen a aquest nucli calcular, i així poder saber quin és l'espai de memòria en el que ha de treballar. La segona etapa ha de calcular les transformacions dels pares seguint l'estructura jeràrquica de l'arbre. I finalment la tercera etapa es fa el càlcul de la transformació final del tronc, i s'escriu al buffer de sortida. El seu funcionament més en detall es pot trobar a la secció *9.3.3 Càlcul de transformacions a nivell d'arbre*.

```
ComputeShader 2:
	IndexEsc = CalcularNumTroncsAnteriors();
	Pare = VectorPares[indexBranca]
	Mentre Pare > -1: No haguem arribat a l'arrel
		TransformacióPare = UltimaPosicioBranca[Pare];
		Pare = VectorPares[Pare]; Pare del pare
	Fi mentre
	Mentre Hi hagi troncs a la branca:
		EscriureBufferSortida(Tpare * Ttronc);
		IndexEsc++; Escrivim a la seguent posicio
	Fi mentre
```

8.6 Diagrama de seqüència Usuari-Aplicació

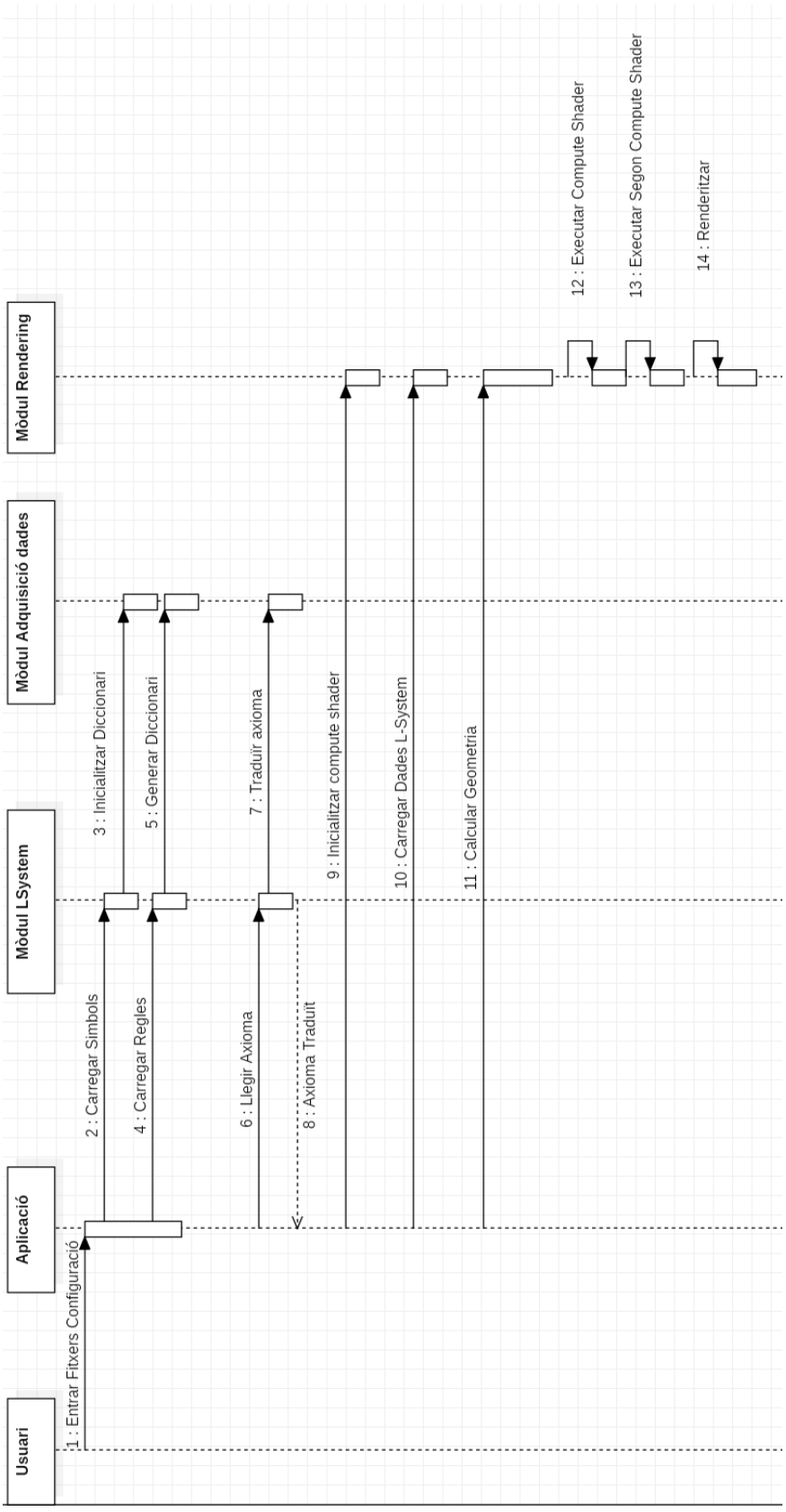


Figura 35 Diagrama de Seqüència Usuari - Aplicació

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

1. L'usuari executa l'aplicació amb els fitxers de configuració establerts.
2. L'aplicació crida la funció de lectura de símbols del mòdul L-System.
3. El mòdul L-System llegeix els fitxers i crida la funció d'inicialitzar el diccionari.
4. L'aplicació crida la funció de llegir fitxer de regles del mòdul L-System.
5. El mòdul L-System llegeix les regles i crida la funció de generació del diccionari del mòdul d'adquisició de dades.
6. L'aplicació crida la funció de llegir el fitxer d'axiomes al mòdul L-System.
7. El mòdul L-System demana la traducció dels axiomes al mòdul d'adquisició de dades (diccionari.
8. El mòdul L-System retorna la traducció a l'aplicació.
9. L'aplicació crida la funció d'inicialització del el mòdul de rendering.
10. L'aplicació crida la funció de càrrega les dades provinents del L-System al mòdul de rendering.
11. L'aplicació crida la funció de calcular la geometria del mòdul de rendering.
12. El mòdul de rendering executa el primer compute shader.
13. El mòdul de rendering executa el segon compute shader.
14. El mòdul de rendering executa la funció de dibuixar.

9. Implementació i proves

En aquesta secció s'explicarà les funcions més rellevants a baix nivell, sobretot ens centrarem amb aquelles relacionades amb el processament de dades per la GPU, ja que són les que tenen més complicació i que no s'han seguit cap referència.. També s'explicarà el perquè s'ha decidit fer d'aquesta manera i els problemes que han anat sorgint.

9.1 Main

En el cas del programa principal trobem dues versions, la versió de la GPU i la versió de la CPU, amb canvis importants ja que tenen relació amb com estan pensats els algorismes dels mòduls.

9.1.1 CPU

Tal i com s'ha dit en la secció *8.4.1 Main,* el programa principal és l'encarregat d'inicialitzar els mòduls de L-System i Rendering, així com proporcionar les dades d'entrada. En el nostre cas, les dades d'entrada és l'axioma inicial, ja sigui llegit d'un fitxer com és el cas, o bé preguntant directament a l'usuari, i l'arbre generat per L-System .

```
int main () {
   string axiom = LLegirAxioma ("Axioms.txt");//Llegir el primer axioma
   LSystem * ls = LSystem::init("Rules.txt");//Inicialitzar Modul LSystem
   ls->NewAxiom(axiom);//Assignar l'Axioma al LSystem
    string tree = ls->GenerateTree (3, false);//Generar amb 3 iteracions
   TreeRenderer tr (tree);//Inicialitzar Modul de render i processar l'arbre
   return 0;
}
```

Codi 1 Programa principal versió CPU

9.1.2 GPU

A diferència del programa de CPU, aquí cal inicialitzar els buffers i organitzar-los. S'ha decidit que sigui l'usuari qui ho faci i que aquest aspecte no quedi amagat dins del mòdul perquè així l'usuari pot decidir generar múltiples buffers i fer diferents boscs. D'aquesta manera li donem tot el control sobre les dades i la flexibilitat per poder decidir com vol emprar les dades. La contrapartida és que l'usuari s'ha de fer càrrec de tota la gestió dels buffers, no només els que

guarden els arbres generats, sinó també els buffers auxiliars com el de matemàtiques. Aquest últim buffer serveix per emmagatzemar les transformacions d'acord amb el diccionari usat per l'usuari, tal i com s'ha explicat a la secció *8.5.4.a Càlcul de transformacions a nivell de branca*.

Així doncs, tal i com es veu al *Codi 2 Programa principal versió GPU* els passos a seguir són:

1. Declarar els buffer d'arbres i d'índex que es vol utilitzar i inicialitzar-los a 0.
2. Establir el nombre de generacions.
3. Inicialitzar el mòdul de L-System, entrant el fitxer de regles que es vol utilitzar.
4. Utilitzar el mòdul de L-System per generar els arbres i guardar-los dins els buffers generats al punt 1.
5. Inicialitzar el mòdul de rendering.
6. Enviar les dades del buffer generades al punt 4.

```
int main(){
   //Generar els buffers: el que guardarà l'arbre, i el que guardarà els indexs
   unsigned short * buffer = new unsigned short[Config::MAX_BUFFER_SIZE];
   unsigned int * indices = new unsigned int[Config::MAX_TREE];

   for (int i = 0; i < Config::MAX_BRANCHES * Config::MAX_WIDTH *
Config::MAX_GENERATIONS; i++) //Posar els buffers a 0
   {
      buffer[i] = 0;
      if (i < Config::MAX_GENERATIONS)
        indices[i] = 0;
   }


//Establim les generacions a 3 i iniciem el mòdul de LSyste
   int generations = 3;
   LSystem * ls = LSystem::init("Rules.txt");
//Generem els diferents arbres i els guardem al buffer. Ens retorna el nombre
d'arbres creats:
   int numTree = ls->GenerateTreesFromFile(buffer, indices, "axioms.txt", generations,
false);
//Comprovar que s'han generat diferents arbres:
   if (numTree < 0)
   {
      throw ("Error generating trees");
   }

// Inicialitzem la classe matemática que ens dóna un buffer amb les matrius de
transformación d'acord amb l'ordre de les operacions establerta pel diccionari
   LSystemMath lsMath;
   glm::mat4 * buffer_dic = new glm::mat4[40]; //Buffer on guardarem les
transformacions
   Dictionary dic;
   lsMath.GenerateMatrixOperationBuffer(buffer_dic, dic, false);


//Agafem la posició d'inici i final del buffer dels arbres.
   int start = 0;
   int end = indices[numTree - 1];

//Mòdul de rendering
//Inicialitzem la clase del compute shader, i enviem la informació
   ComputeShader cs(path, numTree, false);
   cs.SetForBranchCompute(buffer, start, end, buffer_dic, 40, false);
   cs.ComputeBranch();

   return 0;
}
```

Codi 2 Programa principal versió GPU

9.2 LSystem GPU

La següent funció que ens trobem és la de generar els arbres amb el format de buffer per la GPU tal i com s'explica a la secció *8.5.3 LSystem:* .En aquest pas es va baixant el nivell de detall des de començar a nivell d'arbre, fins arribar al nivell de branca. Aquest pas és crucial per poder adaptar tota la informació a les nostres estructures, i si en un futur cal canviar la manera com es guarden les dades, el primer nucli que s'haurà de canviar serà aquest.

9.2.1 Generar Múltiples Arbres

La primera funció de tot aquest nucli simplement llegeix una quantitat d'axiomes d'un fitxer de text i envia el contingut un a un, a que es processi i es generi l'arbre que li correspon. Un cop tenim l'arbre amb totes les seves generacions, únicament ens quedem amb la última de totes ja que en el nostre cas no ens interessen els altres.

Pel que fa la mida dels buffers, són controlats per una classe de configuració que únicament guarda els paràmetres de quins són els límits que permet l'aplicació. Tots els buffers creats en aquesta funció són volàtils, és a dir, s'esborren al final ja que únicament serveix per emmagatzemar la generació de l'arbre de cada axioma i per tant, a cada iteració es canvia el contingut.

Els passos per generar múltiples arbres són (veure *Codi 3 Generació de multiples arbres per GPU*):

1. Declarar els buffers que s'ompliran per cada arbre processat.
2. Llegir els axiomes del fitxer.
3. Per cada axioma.
    a. Generar un arbre a partir de l'axioma i guardar el resultat als buffers generats al punt 1.
    b. Copiar l'última generació de l'arbre anterior al buffer de sortida de la funció *treeBuffer* , i escriure quina és la l'ultima posició que ocupa el l'arbre al buffer d'index *indiceTreeBuffer*.

```
int StringTree::GenerateMultipleToBuffer(unsigned short * Treebuffer, unsigned
       int * indiceTreeBuffer, string filename, Dictionary dic, unsigned int
       generations, bool debug)
{
       //Buffer to store the tree generation
   unsigned short * auxiliarBuffer = new unsigned short[Config::MAX_BRANCHES
       * Config::MAX_WIDTH * Config::MAX_GENERATIONS];

       //Buffer to store the tree generation indices.
   unsigned short * indiceBuffer = new unsignedshort[Config::MAX_GENERATIONS]

   int WrittingPos = 0;
   vector<string> Axioms = ReadAxiomsFromFile(filename);

   //For each axiom:
   for (int axiomIndex = 0; axiomIndex < Axioms.size(); axiomIndex++)
   {
      GenerateToBuffer(auxiliarBuffer, indiceBuffer, Axioms[axiomIndex], dic,
       generations, false);

      //Check reading position and end position
      int start = (generations < 2) ? 0 : indiceBuffer[generations - 2];
      int end = indiceBuffer[generations-1];

      //Start to copy content to auxiliar buffer , to final buffer. Only last
       generation is copied
      for (int i = start; i < end; i++)
      {
       if (WrittingPos < Config::MAX_BUFFER_SIZE)
       {
              Treebuffer[WrittingPos] = auxiliarBuffer[i];
              WrittingPos++;
       }
       else
       {
              cerr << "MAX BUFFER REACHED!" << endl;
       }

      }

   }
   //Clean Buffers
   if (auxiliarBuffer) {
      delete auxiliarBuffer;
      auxiliarBuffer = nullptr;
   }

   if (indiceBuffer) {
      delete indiceBuffer;
      indiceBuffer = nullptr;
   }

   return Axioms.size();
}
```

Codi 3 Generació de multiples arbres per GPU

9.2.2 Generar Arbre

Aquesta funció és l'encarregada de transformar els axiomes en arbres, i per tant, primer de tot ha de tractar l'axioma que arriba en format de caràcters i transformar-lo en format numèric, i al mateix temps adaptar-lo seguint les regles acordades (veure *8.5.2 Format de buffer per L-System*). Un cop s'ha adaptat, es pot tractar que l'axioma sigui una generació prèvia ja que té el mateix comportament (s'estructura igual dins els buffers), i per tant només cal fer el processament d'una nova generació agafant les dades de la generació anterior.

Cal tenir en tot moment els índex d'on acaba i comença una generació dins el buffer. En el cas de l'axioma, la funció que el processa retorna la posició d'acabament. En canvi, en el cas de processar una nova generació, és la mateixa funció que s'encarrega de mantenir els índexs correctes.

Tal i com es pot veure en el *Codi 4 Generació d'un arbre en GPU* es pot observar que el procés de generar un arbre passa pels següents punts:

1. **Inicialitzar la posició d'escriptura del buffer**, és a dir, la primera posició que s'escriurà.
2. **Processar l'axioma**, el qual passa d'una cadena de caràcters a el format correcte numèric i en buffer. El resultat ja es guarda dins el buffer de sortida perquè és considera que és la generació 0.
3. Actualitzar la posició d'escriptura.
4. Per cada generació:
    a. **Processar una nova generació** , retorna si la generació s'ha pogut generar correctament. En cas contrari es para l'execució.

```
void StringTree::GenerateToBuffer(unsigned short * buffer, unsigned short *
       indiceBuffer, string Axiom, Dictionary dic, unsigned int generations ,
        bool debug )
{
   unsigned int nextPositionToWrite = 0;
   //Process Axiom and get it's memory size
   nextPositionToWrite = ProcessAxiom(buffer, nextPositionToWrite,1
      ,Axiom,dic);

   //Store where the axiom finishes
   indiceBuffer[0] = nextPositionToWrite;

   unsigned int currentgeneration = 1;     // Process Tree (Gen 0 == Axiom)
   bool correct = true;

   while(correct && currentgeneration < generations)
   {
      //Process Next Generation
      correct = ProcessNewGeneration(buffer, indiceBuffer, currentgeneration,
       dic, debug) > 0;

      currentgeneration++; //Update generation
   }
}
```

Codi 4 Generació d'un arbre en GPU

9.2.2.a Processar Axioma

Aquesta funció és la responsable de convertir un axioma en format cadena de caràcters, a un axioma numèric i adaptat als buffers. A més a més, també ha de tractar amb possibles errors i aplicar les normes establertes pels buffers (veure la secció *8.5.2 Format de buffer per L-System*).

Per processar l'axioma s'ha de tenir en compta que usualment les branques no estan ben escrites ja que no s'obren i es tanquen degudament. Aquest error no és considera una mala pràctica i tots els sistemes de L-System ha de ser capaços de gestionar-ho.

- **Error:** ABA [C [D] A] C [D]
- **Correcte:** ABA [C [D] [A]] [C [D]]

En aquest cas comprovem que sempre que es tanca una branca, no es pot escriure si no se'n obre una de nova. Per tant, fem el seguiment de si l'estructura és correcta o no.

Un altre detall a tenir en compte és el càlcul de la posició on escrivim del buffer. En comptes de tenir un comptador absolut que indica la posició, fem el càlcul de quina posició toca utilitzant

un comptador relatiu a la branca i un offset que es calcula en funció de les branques que ja tenim i la posició inicial que hem començat a escriure. Aquest sistema es repetirà en diferents funcions.

Tal i com es pot veure al *Codi 5 Processar un Axioma en GPU* els passos per processar un axioma són :

1. **Inicialitzar les variables inicials**, entre elles hi ha el comptador de branques dins de l'axioma *branches*, la profunditat en la que es comença *correntDepth* la qual es passa el seu valor com a paràmetre de la funció. També cal saber on començar a escriure dins el buffer *currentWritablePos*.
2. **Guardar la profunditat inicia**, com que el primer nombre de cada branca indica quin nivell de profunditat es troba la branca, el primer que s'ha d'escriure al buffer de sortida és la profunditat de la primera branca.
3. **Per cada símbol de l'axioma aplicar la seva operació:**
    a. Si el símbol indica que es vol obrir una branca, aleshores cal crear una branca nova, i tancar l'actual. Aquest fet comporta que s'ha d'actualtizar la profunditat actual i la posició d'escriptura al buffer.
    b. En canvi si s'ha de tancar una branca, aleshores simplement cal actualitzar la profunditat actual. Cal tenir present que si es tanca una branca significa que per poder seguir processant l'axioma caldrà obrir-ne una altre.
    c. Si no es cap de les anteriors, aleshores el primer de tot és comprovar que hi hagi una branca oberta, si no hi és, es força a obrir-ne una. Seguidament ja es pot fer la traducció del símbol de caràcter a numèric i guardar-lo a la posició del buffer que toca, sempre hi quan, el símbol existia dins el diccionari.

```
unsigned int StringTree::ProcessAxiom(unsigned short * buffer, unsigned int
      initalPositionWrite,unsigned short initalDepth, string Axiom, Dictionary _dic)
{
   unsigned short branches = 0; // numBranches
   unsigned short currentDepth = initalDepth;//Usually 1
   unsigned int currentWritablePos = initalPositionWrite; //Is there any offset?
   buffer[currentWritablePos] = currentDepth; //Store initial depth
   currentWritablePos++;
   bool brancaCorrecte = true;
   for (int i = 0; i < Axiom.length(); i++)
   {
      if (Axiom[i] == '[') // New Branch
      {
       if (GenerateNewBranch(buffer, &branches, &currentWritablePos, &currentDepth,
initalPositionWrite))
              brancaCorrecte = true;
       else
              return 0; //Could not create a new branch, probably run out of space!
      }
      else if (Axiom[i] == ']') // Close Branch
      {
       currentDepth-=1;
       brancaCorrecte = false;
      }
      else // Populate Branch
      {
       if (currentWritablePos < Config::MAX_WIDTH) //If there is enough space
       {

              if (!brancaCorrecte)//If there isn't any open branch = open new one
              {
                     if (GenerateNewBranch(buffer, &branches,
              &currentWritablePos, &currentDepth, initalPositionWrite))
                            brancaCorrecte = true;
                     else
                            return 0;//Could not create a new branch
              }
              //Translate char to short
              unsigned short item = _dic.TranslateCharToShort(Axiom[i]);
              if (item != 0) //Escrivim al buffer
                     buffer[branches * Config::MAX_WIDTH + initalPositionWrite +
currentWritablePos] = item;
              currentWritablePos++;
       }
      }
   }

   return (branches+1) * Config::MAX_WIDTH + initalPositionWrite;
}
```

Codi 5 Processar un Axioma en GPU

Generar Branca

Una funció associada al processament de l'axioma i que trobarem més endavant és la de **GenerateNewBranch()**, la qual ens prepara els índex d'escriptura a la posició del buffer on s'hauria de començar a escriure la nova branca. Al mateix temps insereix la profunditat que li pertoca tal i com es pot veure al *Codi 6 Generar una nova branca*.

El funcionament és el següent:

1. **Incrementar el nombre de branques i de profunditat**.
2. **Comprovar que hi ha suficient espai al buffer** per poder escriure una nova branca.
3. **Escriure a la posició del buffer que li pertoca la nova profunditat**. Tal i com s'ha esmentat en l'apartat 9.2.2.a Processar Axioma l'índex d'escriptura, *currentWritablePos*, és un índex relatiu a la branca, i per tant anirà de 0 – 15 (sabent que la branca té 16 posicions). Això fa que la posició on s'ha de començar a escriure una branca s'hagi de calcular de forma absoluta:

$$posició = numBranques * AmpladaBranca + offsetIncial$$

```
bool StringTree::GenerateNewBranch(unsigned short * buffer, unsigned short *
   branches, unsigned int * currentWritablePos, unsigned short * currentDepth,
   unsigned int initalPositionWrite)
{
   *branches+=1;
   *currentDepth+=1;
   //Check if there is space for the new branhc
   if (*branches * Config::MAX_WIDTH + initalPositionWrite > Config::MAX_BUFFER_SIZE)
      return false;

   buffer[*branches * Config::MAX_WIDTH + initalPositionWrite] = *currentDepth;
   *currentWritablePos = 1;
   return true;
}
```

Codi 6 Generar una nova branca

9.2.2.b Processar Nova Generació

La funció de processar una nova generació s'encarrega de llegir l'ultima generació de l'arbre i aplicar-li el L-System per crear-ne la següent. Malgrat tot la seva tasca principal és la de comprovar la informació de l'última generació, preparar els índex dels buffers i després derivar la feina de processament de cada branca a una funció de més baix nivell. Pel que fa els índexs, en tindrem dos, un de lectura i un d'escriptura: l'índex de lectura sempre apuntarà el principi de la branca que volem processar, mentre que el d'escriptura el gestionarà la funció que s'encarregui de processar les branques, però sempre, acabarà apuntant al inici d'una nova branca, deixant-lo apunt per poder processar una altre branca.

```
unsigned int StringTree::ProcessNewGeneration(unsigned short * buffer, unsigned short
   * indiceBuffer, unsigned int currentGeneration, Dictionary _dic, bool debug)
{
   //Prepare the indices
   unsigned int num_branches = 0;
   unsigned int writtingPos = 0, readingPos = 0;

   //Indice where to start to read
   (currentGeneration < 2) ? readingPos = 0 : readingPos =
indiceBuffer[currentGeneration - 2];
   //Indice where to start to write
   writtingPos = indiceBuffer[currentGeneration - 1];
   //Check the branches inside the last generation
   num_branches = (writtingPos - readingPos) / Config::MAX_WIDTH;

   //For each branch:
   for (int i = 0; i < num_branches; i++)
   {
      writtingPos = ProcessBranch(buffer, readingPos, writtingPos, _dic, debug);
      if (writtingPos == 0)
      {
       return 0;
      }
      //Update the reading position by the width offset.
      readingPos += Config::MAX_WIDTH;
   }
   //Set the final position of the current generation
   indiceBuffer[currentGeneration] = writtingPos;
   return writtingPos;
}
```

Codi 7 Processar una nova generació

Processar Branca:

La funció processar branca és molt semblant a la funció de **GenerarBranca()** (Veure *Generar Branca*), la diferència més important és que ara no cal passar d'una cadena de caràcters a numèriques i adaptar-ne el format, sinó que ara ha de llegir del mateix buffer, la branca de la generació anterior, i aplicar-li el L-System.

Una altre diferència destacable és que ara caldrà traduir els símbols de la generació anterior amb les regles que li pertoquen, i es pot donar el cas que una regla tingui branques a dins, el qual ens porta al mateix problema que els axiomes, podria ser que la gramàtica no fos correcta.

El codi es pot trobar a l'annex (

Document 1. Codi de la funció Processar Branca).

El resultat d'aplicar l'operació és tenir tota una nova generació dins del buffer d'entrada, i el buffer d'índexs actualitzats.

9.3 Mòdul de rendering GPU

En el següent apartat ja entrem dins el motor de rendering, on les dades que necessita és el buffer d'arbres que ha de renderitzar. Aquest buffer l'hem construït en l'apartat *9.2.1 Generar Múltiples Arbres*.

El primer pas és carregar la geometria, tal i com s'explica en la secció *8.4.4.a Afegir geometria* ,concretament en l'apartat *Inicialització i càrrega dels models* . D'aquest apartat entrarem en detall sobretot a la funció de normalització del model, que és important per després a la hora de dibuixar es vegi correctament.

9.3.1 Normalització dels models

Recordem que quan carreguem el model hem de llegir tots els vèrtex i aprofitem aquest moment per determinar quin és la coordenada mínima i màxima per cada eix. I finalment, determinem quin és l'eix més gran de tots fent la resta entre el seu màxim i mínim respectiu. D'aquest resultat en diem **nomFactor**. Amb aquestes dades ja som capaços de normalitzar el nostre model fent que totes les coordenades estiguin compreses entre 0-1. Però amb això no n'hi ha prou, ja que després el centre de l'objecte serà un punt entre (0.0,0.0,0.0) i (1.0,1.0,1.0), i per poder fer les transformacions de manera segura és bo tenir el centre al punt (0.0,0.0,0.0). Per això cal calcular el centre del objecte aplicant la normalització i després desplaçar tots els vèrtex el valor d'aquest centre.

Aquesta operació també ens protegeix del fet que el model tingui per defecte el pivot point a un lloc correcte, ja que a vegades en els programes 3D els models no estan centrats al (0,0,0). Així nosaltres el recalculem i ens assegurem que estigui tot al seu lloc.

```
void model::normalizeVertexAxis(vector<Vertex> &v, float minX, float minY, float
    minZ, float normFactor,float max_x,float max_y,float max_z ,bool debug)
{

   glm::vec3 Centre(
      (max_x - minX)/(2*normFactor) ,
      (max_y - minY )/ (2 * normFactor),
      (max_z - minZ)/ (2 * normFactor)
   );

   for (int i = 0; i < v.size(); i++)
   {
      // Normalize between 0 - 1
      v[i]._posicio.x = ((v[i]._posicio.x - minX) / normFactor) ;
      v[i]._posicio.y = (v[i]._posicio.y - minY) / normFactor;
      v[i]._posicio.z = (v[i]._posicio.z - minZ) / normFactor;

      // Move the model to make the center to (0,0,0)
      v[i]._posicio.x -= Centre.x;
      v[i]._posicio.y -= Centre.y;
      v[i]._posicio.z -= Centre.z;
 }
```

Codi 8 Normalització d'un model 3D

9.3.2 Càlcul de transformacions a nivell de branca

El càlcul de transformacions a nivell de branca es fa ja amb el buffer del L-System generat, tal i com es mostra en l'apartat *9.2.1 Generar Múltiples Arbres*, i es calcula utilitzant el compute shader. Quan s'executa el compute shader, primer s'ha de preparar els buffers, s'han de compilar els shaders i també cal indicar quants work groups (veure *5.1.5.c Arquitectura per Compute Shader*) es volen utiltizar. En el nostre cas s'utilitzaran tants workGroups (de 1 sol nucli) com branques tinguem. Per saber el nombre de branques, tenim el buffer d'índexs, que diu on comença i on acaba cada arbre, per tant, només cal fer :

$$nBranques = \frac{Final - Inicial}{Amplada\_de\_Branca}$$

L'amplada de branca està definida a la classe ***Config***, amb un valor que serà de 16 ja que és el que hem definit. Es pot trobar tot el codi respectiu a la preparació dels buffers i el càlcul de branques a l'annex: *Document 2. Codi per preparar els buffers d'un compute shader* i *Document 3. Operació per executar compute Shader*.

Entrant al contingut del shader que és l'encarregat del processament de les dades, tenim dos inputs i un output:

- El buffer **branches** que correspon als arbres generats per LSystem.
- El buffer **transforms** que són les transformacions per cada operació.
- El buffer **outs** que és on guardem les dades de sortida.

El primer que cal fer és determinar quin workgroup s'està executant. La comanda (*gl_WorkGroupID.x*) ens diu l'índex del workgroup, que equival a la branca que li pertoca llegir.

Per tant, la posició on ha de començar a llegir és: $pos = índex * midaBranca$ que en el nostre cas és 16. Com a notació, la mida de la branca està posada com a constant fixa, i per tant, si mai canviem la mida, haurem de canviar la constant, o bé passar-la com una variable des de la CPU.

Després cal recórrer tota la branca aplicant la transformació que li toca, recordem que el símbol (numèric) també és l'índex de la transformació que cal buscar dins el buffer de transformacions. Per tant, tenim un accés directe, fet que optimitza molt el procés. Sempre que trobem un tronc, l'hem d'afegir a la matriu de sortida.

```
#version 430 core
#extension GL_NV_gpu_shader5 : enable //Enables computeShader

layout (local_size_x = 1) in;
layout(binding=0) buffer inputBuffer {
    uint16_t branches[];
};
 layout(binding = 1) buffer transformBuffer{
   mat4 transforms[];
 };
layout(binding=2) buffer outputBuffer {
    mat4 outs[];
};

void main()
{
   const uint MAX_TRONCS = 16;
   const uint index =  (gl_WorkGroupID.x) * 16;

   mat4 ScaleMatrix = mat4(0.9);
   ScaleMatrix[3][3] = 1;
   mat4 currentMatrix = mat4(1.0);
   mat4 lastMatrix = currentMatrix;
   uint troncs = 1;

   int i = 1;
   while(i < 16 && troncs < MAX_TRONCS-1)
   {
      //Update current matrix
      currentMatrix = currentMatrix * transforms [ input[index + i] ];
      //If the symbol = trunk (tronc)
      if(branches[index + i] == uint16_t(3) || branches[index + i] == uint16_t(4))
      {
        currentMatrix = currentMatrix * ScaleMatrix;
      //Write the transform to the buffer
        outs[gl_WorkGroupID.x * MAX_TRONCS + troncs] = currentMatrix;
        lastMatrix = currentMatrix;
        troncs ++;
      }
      i++;
   }
   // First matrix of outputBuffer = number of trunk in the branch
   outs[gl_WorkGroupID.x * MAX_TRONCS] = mat4(troncs - 1);

   while(troncs < MAX_TRONCS) //Fill with identity
   {
      outs[gl_WorkGroupID.x * MAX_TRONCS + troncs] = mat4(0.0);
      troncs ++;
   }
   //Last Matrix of outputBuffer = the transform of the last trunk
   outs[gl_WorkGroupID.x * MAX_TRONCS + MAX_TRONCS-1 ] = lastMatrix;
}
```

Codi 9 ComputeShader geometria a nivell de branca

9.3.3 Càlcul de transformacions a nivell d'arbre

Per calcular les transformacions a nivell de d'arbre cal saber quina branca penja de quina per poder fer tot l'arbre de transformacions i dependències. És per aquest motiu que el primer pas és un pre-processat dins de la CPU que determina la jerarquia de branques. Aquest pre-processat es pot fer en qualsevol moment ja que tenim les dades provinents del mòdul de L-System, però s'ha decidit fer-ho en aquest moment perquè és el moment on que necessitem aquestes dades.

L'algorisme que fa aquest processament està dividit en dues parts, la primera serveix per determinar la jerarquia de totes les branques del buffer, independentment de l'arbre. La segona determina quines branques són d'un arbre o d'un altre.

9.3.3.a Determinació de la Jerarquia

Per determinar la jerarquia primer cal fer una ullada a com està el buffer. El primer element de cada branca indica el nivell de profunditat del qual depèn, per tant, l'arrel és el nivell 1 (El nivell 0 es reserva).

Aleshores s'utilitza un buffer per guardar quina era l'última branca que tenia una profunditat determinada. Els nivells de profunditat per defecte s'inicialitzen a -1, indicant que no hi h cap branca en aquell nivell, fet que farà que el nivell 0 sempre hi hagi un -1, ja que les arrels comencen al nivell 1. Veure *Figura 36 vector que emmagatzema a cada nivell de profunditat l'última branca*.

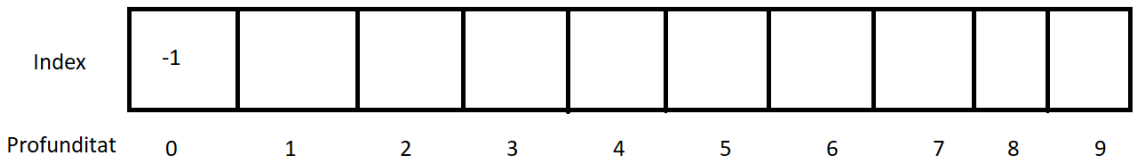


Figura 36 vector que emmagatzema a cada nivell de profunditat l'última branca.

Com que el buffer del L-System on guardem les branques dels arbres es genera seguint l'ordre de creació de les branques, aleshores sempre que busquem la branca més propera amb un nivell de profunditat inferior, serà el pare de la branca.

```
void ComputeShader::RetrieveParents()//RedoForMultipleTree
{

   short *auxiliar = new short[_buff.num_branches+1]; // Used to update the last
branch on the depth level

   for (unsigned int i = 0; i < _buff.num_branches; i++)//init buffer at -1
      auxiliar[i] = -1;

   for (unsigned int branch = 0; branch < _buff.num_branches; branch++)
   {
      // Get the depth level of the branch
      unsigned short nivell = _buff.inputBuffer[Config::MAX_WIDTH * branch];

      //update last branch at that level
      auxiliar[nivell] =  branch;

      //Retrieve the lower depth branch
      _buff.parentBufferIndices[branch] = auxiliar[nivell - 1];

   }

   delete auxiliar;
}
```

9.3.3.b Determinació dels arbres

Un cop tenim els pares de cada branca determinats, hem de saber quines branques pertanyen a quin arbre, i per fer-ho necessitem contar quantes branques té cada arbre. Al compute shader anterior (veure secció *9.3.2 Càlcul de transformacions a nivell de branca*) guardem al principi de cada branca una matriu indicant el nombre de troncs que hi pertanyen. Amb aquesta informació a més de que sabem la jerarquia de totes les branques i que totes les arrels tindran un -1 com a índex de branca pare, podem determinar on comença i on acaba cada arbre. *Veure Codi 10 Càlcul de la jerarquia de troncs*.

```
void ComputeShader::RetrieveParentsAndTrees()
{
   int branch = 0;
   int tree_num = -1;
   while (branch < _buff.num_branches)
   {
      if (_buff.parentBufferIndices[branch] == -1)//If he has no parent == root
        tree_num++;

      if (tree_num > -1)//Check that atleast there is one tree
       //get the position[0][0] of the matrix = num of trunk inside the tree
       _buff.treeChunksCount[tree_num]+=
              _buff.computedBuffer[Config::MAX_WIDTH*branch][0][0];

      branch ++;
   }

}
```

Codi 10 Càlcul de la jerarquia de troncs

9.3.3.c Càlcul de les transformacions

Finalment, amb totes les dades processades, ja estem apunt per utiltizar el segon computeShader, que agafa les dades de les transformacions dels troncs dins de la branca (Veure secció *9.3.2 Càlcul de transformacions a nivell de branca*), conjuntament amb la informació jeràrquica de les branques i calcular les transformacions relatives a l'arbre.

L'algorisme consta de tres parts (*veure Codi 11ComputeShader càlcul dels troncs a nivell d'arbre*):

1. Calcular quants troncs hi ha abans de la branca que està processant, així trobem l'espai de memòria del buffer que es pot escriure.
2. Calcular les transformacions acumulades de les branques pare, seguint l'ordre jeràrquic que ve donat pel buffer de pares.
3. Calcular la transformació final dels troncs.

```
#version 430 core
#extension GL_NV_gpu_shader5 : enable

layout (local_size_x = 1) in;

layout(binding=0) buffer branchBuffer {
    mat4 branch[];
};

layout(binding = 1) buffer parentBuffer{
   int16_t parent[];
 };

layout(binding=2) buffer outputBuffer {
    mat4 outs[];
};

void main()
{
   const uint BRANCH_OFFSET = 16;
    const uint index =  (gl_WorkGroupID.x) * BRANCH_OFFSET;
   const uint MAX_BRANCHES = 14;

   int indexWrtiteOffset = 0;
   for(int i = 0; i < gl_WorkGroupID.x; i++) // Calculate where should start to write
   {
      indexWrtiteOffset = indexWrtiteOffset + int(branch[i * BRANCH_OFFSET][0][0]);
   }


   mat4 parentTransform = mat4(1.0);
   int nextParent = parent[gl_WorkGroupID.x];
   while(nextParent >= 0) // Calculate parents transforms
   {
      parentTransform =  branch[(nextParent+1) * BRANCH_OFFSET - 1] * parentTransform;

      nextParent = parent[nextParent];
   }


   int indexBrancaEscriptura = 0;

   while(indexBrancaEscriptura < branch[index][0][0])
   {
      outs[indexWrtiteOffset + indexBrancaEscriptura] = parentTransform *branch[index
+indexBrancaEscriptura+1];
      indexBrancaEscriptura++ ;
   }

}
```

Codi 11ComputeShader càlcul dels troncs a nivell d'arbre

9.3.4 Càlcul de les transformacions absolutes

Un últim pas i que no s'ha esmentat anteriorment és el pas de les transformacions calculades fins el moment que són relatives a l'arbre, a les transformacions del món on es veuran. Aquest pas es fa utilitzant el vèrtex shader del render, on s'envien com a dades la posició on ha d'aparèixer l'arbre (posició de l'arrel), i la transformació dels troncs.

La posició de l'arrel es calcula sobre una graella de 10 x n, sent n el nombre de columnes necessàries per poder renderitzar tots els arbres. Aquesta col·locació té un offset aleatori per donar una sensació que els arbres no estan alineats els uns amb els altres:

```
//Els roots haurna de ser llegits per un fitxer
   srand(time(NULL)); //init seed
   vector<glm::mat4> roots;
   roots.resize(Config::MAX_TREE, glm::mat4(1.0f));//resize root buffer
   for (int i = 0; i < Config::MAX_TREE; i++) //or each tree
   {

      float offset_x = rand() % 5 - 2.5; //x_jitter
      float offset_z = rand() % 5 - 2.5;z//z_jitter

      int posicio_x = i % 10; //Calcualte grid position x
      int posicio_z = i / 10;//Calcualte grid position z

      //Calculate world position (grid * offset + jitter)
      roots[i][3][0] = posicio_x * 5 - 10/2*5 + offset_x;
      roots[i][3][2] = posicio_z * 5 - 10 / 2 * 5 + offset_z;
   }
```

Codi 12 Plantació dels arbres amb jitter

Per indicar la posició de l'arrel estem utilitzant una transformació de translació que simula el moviment del punt (0,0,0) al punt (x,y,z). Per aquest motiu tenim una matriu 4x4 i no un vector. Si recordem les matrius de translació (veure *5.4.1 Translació*), veiem que els índex que estem sobreescrivint dins de la funció corresponent als índex de desplaçament en l'eix X i eix Z.

Amb les transformacions de les arrels, ja podem calcular les transformacions absolutes. Com que les transformacions dels troncs estan totes seguides dins del mateix buffer, es necessari saber quines branques pertanyen a quin arbre. Per aquest motiu s'utilitza el buffer que s'ha calculat a la secció *9.3.3.b Determinació dels arbres,* on es guarden quants troncs té cada arbre.

```
void renderSystem::RenderInstancingCubes(vector<short> bufferTreeChunk , vector
<glm::mat4>RootTransforms)
{
   while (glfwWindowShouldClose(mainWindow) == false) {
      if (glfwGetKey(mainWindow, GLFW_KEY_ESCAPE) == GLFW_PRESS)
       glfwSetWindowShouldClose(mainWindow, true);
      cameraInput();
      // Background Fill Color
      glClearColor(0.05f, 0.05f, 0.20f, 1.0f);
      glClear(GL_COLOR_BUFFER_BIT);

      int current_tree = 0;//Tree counter
      int chunks = 0; //Trunk counter
      for (int i = 0; i < _elements.size(); i++)//For each trunk
      {

        if (chunks > bufferTreeChunk[current_tree]-1)Check if need to change tree
        {
               current_tree++; chunks = 0;//Get to the next tree and reset counter
        }

        sendUniforms(_elements[i]->get_program(),_transforms[i],
               RootTransforms[current_tree]);
        _elements[i]->draw();
        chunks++;

      }
      // Flip Buffers and Draw
      glfwSwapBuffers(mainWindow);
      glfwPollEvents();
   }
   glfwTerminate();
}
```

Codi 13 Renderització dels troncs

Al vèrtex shader rep la transformació del tronc, la de l'arrel i calcula la posició final:

```
#version 400 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

uniform mat4 view; //Camera view transform
uniform mat4 model; //Camera model transform
uniform mat4 projection;//Camera Projection transform
uniform mat4 transform;//Trunk transform
uniform mat4 root; //Root Transform

out vec3 normal;
out vec4 pos;
void main()
{
   mat4 MVP = projection * view * model;
   vec4 p = root *transform * vec4(aPos,1.0);
   gl_Position =  MVP * p;//Camera transfrom + object transform
   normal = aNormal;
   pos = p;
}
```

Codi 14 Vèrtex Shader

10. Resultats

Resultats en funció dels objectius

L'objectiu d'aquest treball és desenvolupar una generació procedural d'arbres en GPU i el més proper a temps real possible, però per arribar a aquest objectiu primer calia assolir objectius menors. El primer de tot és aconseguir un prototip de generació en CPU que renderitzi un arbre. Aquest objectiu és compleix en el moment que obtenim un L-System complert, com és el cas de la *Figura 37 Arbre generat per CPU*.:

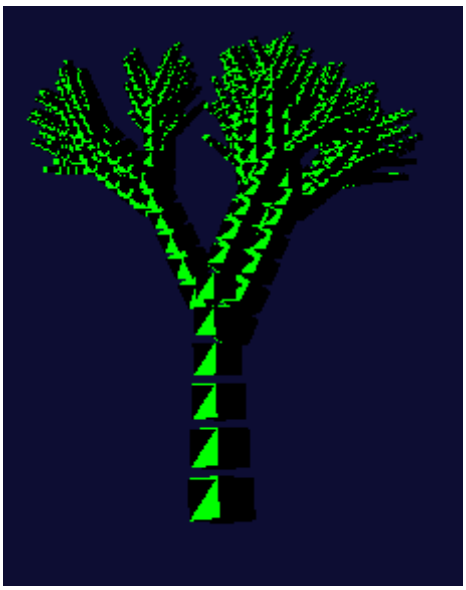


Figura 37 Arbre generat per CPU

Cal recordar que la qualitat del render no és un dels objectius proposats, i per tant, ens conformem amb poder visualitzar l'arbre com una interpretació del processament de les dades.

Un objectiu secundari és el fet de poder carregar tot tipus de model, un exemple és la generació de l'arbre utilitzat el model d'una torre. (*Figura 38 Arbre generat amb GPU amb model alternatiu i Figura 39 Model 3D d'una torre)*.

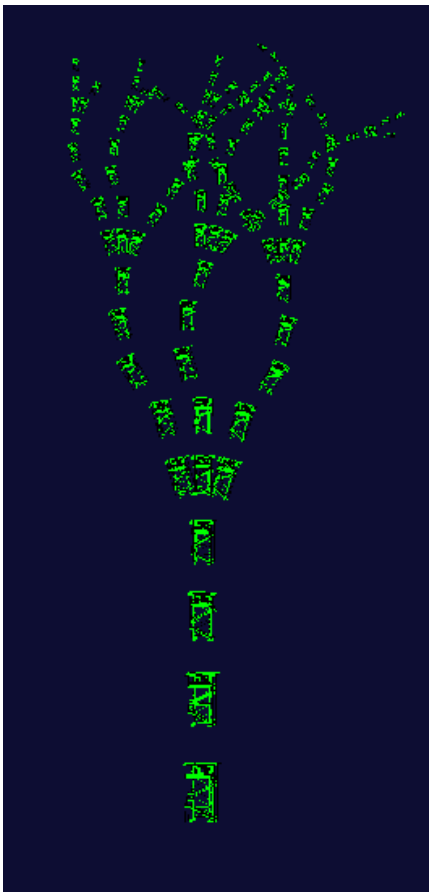


Figura 38 Arbre generat amb GPU amb model alternatiu

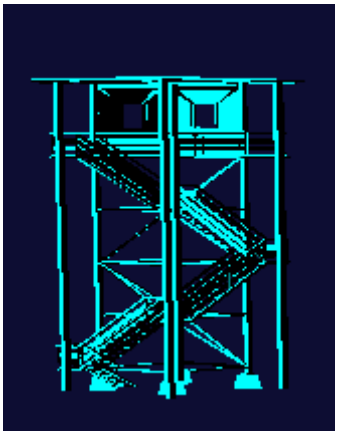


Figura 39 Model 3D d'una torre

Ara ja si entrant dins de la generació procedural per GPU, el primer pas era aconseguir renderitnzar un arbre generat amb el programa de tarja gràfica *Figura 40Arbre Generat en GPU*.

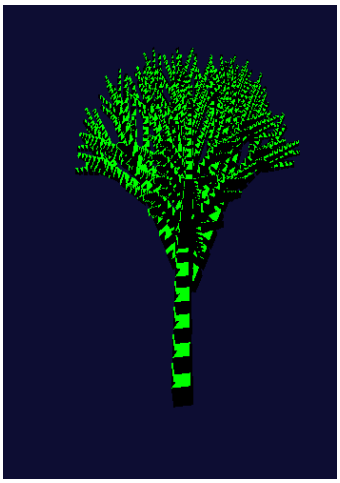


Figura 40Arbre Generat en GPU

Finalment, l'objectiu final consta en generar múltiples arbres en GPU, en aquest cas el renderitzat de diferents grups d'arbres, tal i com es pot veure en les imatges: la primera de totes consta d'un bosc de 100 arbres col·locats de forma arrenglerada (veure *Figura 41 100 Arbres renderitzats per GPU)*. La segona és una representació de 25 arbres amb la tècnica de jitter, la qual causa la il·lusió de que els arbres són plantats aleatòriament (veure la *Figura 42 25 Arbres renderitzats per GPU amb tècnica de jitter*). Finalment la tercera imatge és un bosc de 100 arbres amb la tècnica de jitter (veure la *Figura 43 100 Arbres renderitzats per GPU amb tècnica de jitter).*

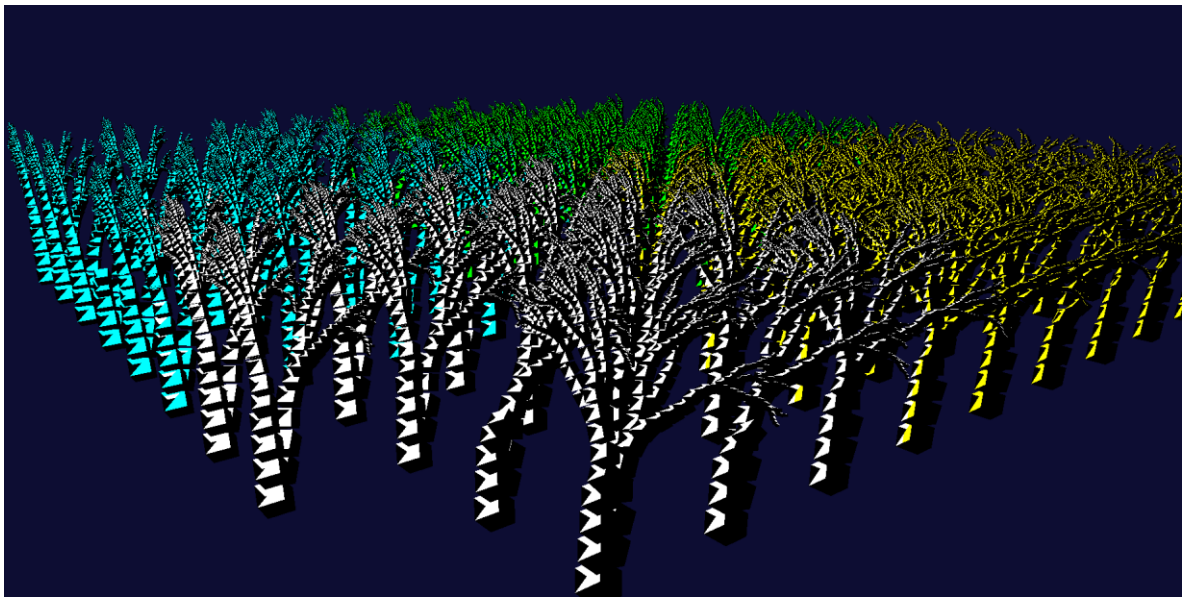


Figura 41 100 Arbres renderitzats per GPU

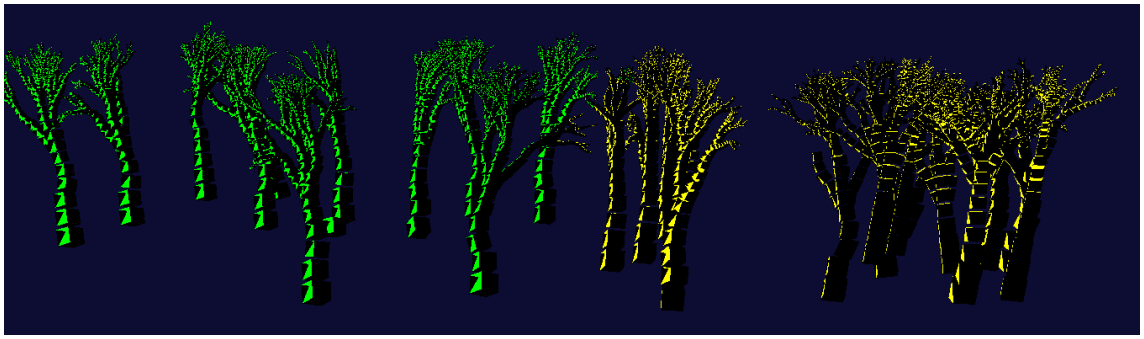


Figura 42 25 Arbres renderitzats per GPU amb tècnica de jitter

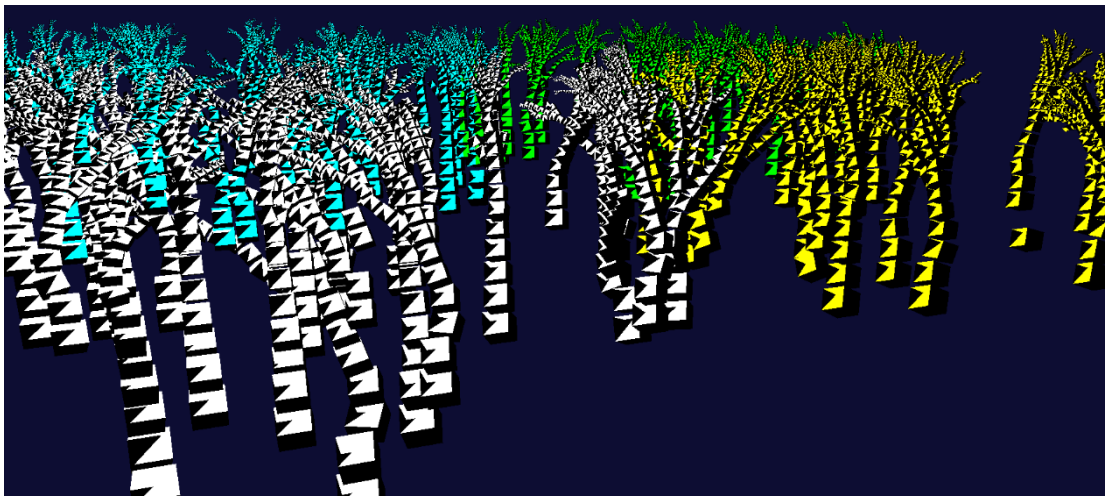


Figura 43 100 Arbres renderitzats per GPU amb tècnica de jitter

Temps de computació:

S'ha decidit no comparar els dos programes de CPU i GPU ja que un està dissenyat per aprendre les bases dels L-Systems i només renderitza un arbre, i el programa de GPU s'ha pensat per renderitzar un grup d'arbres maximitzant la velocitat de rendering. Així que centrarem tot aquest anàlisis en el programa de GPU que des del primer moment ha set l'objectiu d'aquest treball.

Un aspecte important per determinar si hem assolit els objectius es determinar el temps de computació dels arbres, així que hem mesurat dos temps, la part de CPU que processa el L-System, i la part de GPU que processa tota la geometria. Tal i com es pot veure a la *Taula 4 Comparativa de temps de CPU i GPU en funció de la quantitat d'arbres*, tenim uns temps de computació realment baixos, sent gairebé 1s per 100 arbres. El que crida més l'atenció és el temps de la GPU, el qual no sembla que sigui afectat pel nombre d'arbres, mentre que en el cas de la CPU es veu un increment clar (veure *Figura 44Grafic comparativa entre CPU i GPU*)

Num arbres	Temps CPU	Temps GPU	Temps total
1	*0.138s*	*0.072s*	*0.21s*
25	*0.540s*	*0.068s*	*0.608s*
50	*0.784s*	*0.081s*	*0.875s*
100	*0.909s*	*0.087s*	*0.996s*

Taula 4 Comparativa de temps de CPU i GPU en funció de la quantitat d'arbres

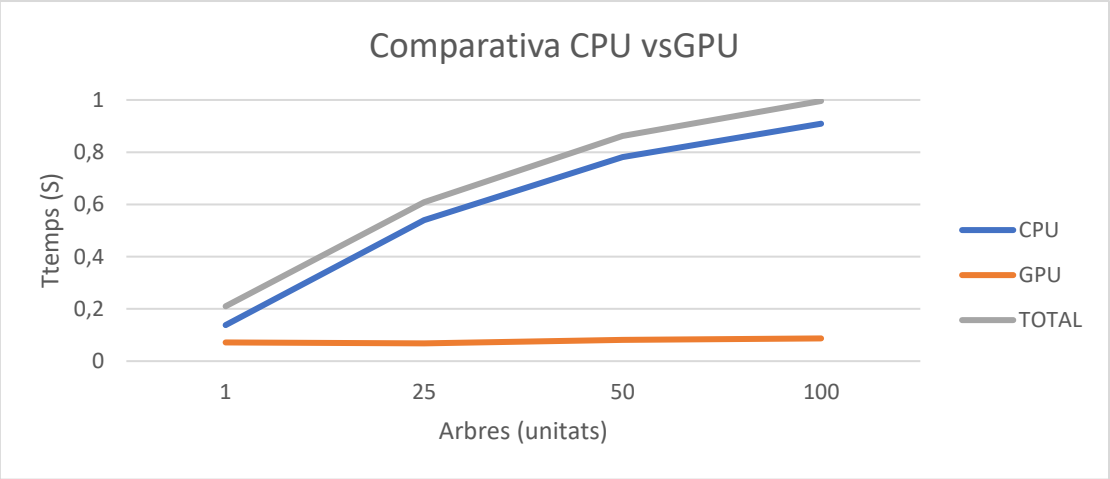


Figura 44Grafic comparativa entre CPU i GPU

Validesa legal de l'aplicació

Un aspecte a tenir en compte quan desenvolupem una aplicació és la validesa legal, és a dir, si compleix la legislació actual pel que fa a el Reglament General de Protecció de dades (RGPD) i la Llei de Serveis de la Societat de la Informació i Comerç Electrònic (LSSICE). Si parlem de la de protecció de dades, l'aplicació no emmagatzema cap tipus d'informació de l'usuari, motiu pel qual complim l'esmentat pel RGPD. Per altra banda, si ens centrem en el comerç electrònic, l'aplicació d'aquest projecte no inclou cap servei de pagament, motiu pel qual complim l'esmentat per la LSSICE.

11. Conclusions

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

S'ha complert i amb escreix l'objectiu d'assolir un bon coneixement en l'àmbit de les targes gràfiques, des de l'arquitectura passant per algunes de les llibreries més utilitzades com OpenGl, fins a preparar tota una estructura de dades i programes pensats exclusivament per ser executats dins d'una tarja gràfica.

S'han obert portes a una forma de càlcul geomètric a la tarja gràfica que poden donar lloc a aplicacions interessants, ja siguin creant arbres, com és el nostre cas, o bé altres aplicacions que requereixen jerarquies geomètriques, com per exemple un esquelet.

Respecte a l'aplicació final, el resultat és bo si tenim en compte que al principi no hi havia res, però això no treu que hi hagi un gran camp de millora, sobretot en fer-ho més agradable a l'usuari, i permetre més opcions i més característiques que ja ofereixen altres programes de generació procedural amb L-Systems.

Com a reflexió final s'han obtingut resultats bons, tot i els problemes i les incerteses de no saber si el que s'estava fent funcionaria. S'ha aconseguit una aplicació procedural en tarja gràfica que fins ara ningú no havia publicat mai en la literatura de l'àmbit, malgrat que no s'ha aconseguit fer tot el procediment dins de la tarja gràfica.

12. Treball futur

Com s'ha comentat a les conclusions aquest projecte està molt lluny de les prestacions que donen altres programes dedicats a la generació procedural de vegetació, però hi ha algunes d'aquestes prestacions que es poden afegir amb poc esforç:

- **Aleatorietat:** una peça clau per a les generacions procedurals és l'aparició d'aleatorietat, i una forma molt senzilla d'afegir és permeten que un símbol tingui més d'una regla. Aleshores cada cop que es busqui la regla associada a un símbol es retornarà una de les possibles seguint una funció d'aleatorietat.
- **Aleatorietat en GPU:** un pas més en l'aleatorietat és poder canviar la jerarquia dels arbres. Si recordem a la secció *9.3.2 Càlcul de transformacions a nivell de branca*, tenim totes les transformacions relatives a una branca, i per tant no hi ha res que ens obligui seguir l'estructura jeràrquica establerta durant el L-System.
- **Millora del render:** en aquest projecte no s'ha posat l'accent en la visualització dels resultats ja que no era una part important, així que hi ha molt espai de millora, i el més important de tot, la base ja està preparada amb la càrrega de textures i processament de malles de vèrtex.
- **Terreny i la plantació d'arbres:** els arbres es planten en una graella flotant, una millora important seria la possibilitat de crear un terreny de forma procedural i plantar-hi arbres a sobre, permeten a l'usuari decidir a quina zona vol que hi hagi vegetació.
- **Gestor de paquets:** actualment s'ha de limitar el nombre d'arbres generats per no desbordar la GPU amb masses dades de cop i saturar els nuclis, però una millora important seria tenir un gestor que empaqueta els arbres que s'han de processar per tal de repartir la càrrega de treball de la tarja gràfica.

Bibliografia

Krhonos. (2000). *Krhonos*. Recollit de https://www.khronos.org

Markus Lipp, P. W. (2009). Parallel Generation of L-Systems. *Vision Modeling and Visualization* .

Muijden, J. V. (2017). *Guerrilla-Games.* Recollit de https://www.guerrilla-games.com/read/gpu-based-procedural-placement-in-horizon-zero-dawn

NIV_ANTERU. (07 / Juliol / 2018). *anteru.net*. Recollit de https://anteru.net/blog/2018/intro-to-compute-shaders/

Przemyslaw Prusinkiewicz, A. L. (2004). *The Algorithmic Beauty of Plants.*

Vries, J. d. (2014). *Learn Opengl*. Recollit de https://learnopengl.com

13. Manual d'usuari i instal·lació

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

13.1 Manual d'usuari

Abans d'executar l'aplicació cal definir els fitxers d'entrada de dades que n'hi ha tres:

- **Símbols:** permet entrar els símbols que vol l'usuari, com a mínim n'hi ha d'haver un per cada operació bàsica.
- **Regles:** permet l'usuari establir quines regles vol per a cada símbol.
- **Axiomes**: permet a l'usuari introduir les arrels dels arbres i el nombre d'arbres que vol calcular.

Es poden trobar els detalls del format de cada fitxer a la secció *8.4.2 Mòdul Adquisició de dades.*

Aquests fitxers es trobaran al directori de l'aplicació a la vista de l'usuari.

El següent pas obrir l'aplicació on se l'hi preguntarà el nombre de generacions i el nom del fitxer on hi ha el model a carregar. Aquest fitxer del model cal que també existeixi en el directori de l'aplicació.

Un cop introduïts només cal esperar a que s'obri la finestra de visualització, aleshores es pot controlar la càmera amb les tecles:

- **W** Moure amunt.
- **S** Moure avall.
- **A** rotar a l'Esquerra.
- **D** rotar a la dreta.
- **Q** Aproximar-se
- **E** Allunyar-se.

14. Annex

Document 1. Codi de la funció Processar Branca

Document referenciat en l'apartat *9.2.2.b Processar Nova Generació*.

Aquesta funció és la responsable de processar una nova generació d'una branca. A més a més, també ha de tractar amb possibles errors i aplicar les normes establertes pels buffers (veure la secció *8.5.2 Format de buffer per L-System*).

S'ha de tenir en compta que les regles entrades per l'usuari poden no estan ben escrites ja que les branques que en puguin derivar no s'obren i es tanquen degudament. Aquest error no és considera una mala pràctica i tots els sistemes de L-System ha de ser capaços de gestionar-ho.

- **Error:** ABA [C [D] A] C [D]
- **Correcte:** ABA [C [D] [A]] [C [D]]

En aquest cas comprovem que sempre que es tanca una branca, no es pot escriure si no se'n obre una de nova. Per tant, fem el seguiment de si l'estructura és correcta o no.

El procediment serà senzill, caldrà comprovar si el símbol equival a obrir o tancar branca i aplicar l'operació corresponent, o bé, si el símbol té una regla associada, que aleshores caldrà substituir-lo per la seva regla. Pot ser que aquesta regla tingui branques a dins, i per tant, caldrà separar les branques de la mateixa forma que s'ha fet en el processament dels axiomes (veure *9.2.2.a Processar Axioma*).

```
unsigned int StringTree::ProcessBranch(unsigned short * buffer, unsigned int
readingBufferPos, unsigned int writtingBufferPos, Dictionary _dic, bool debug)
{
   unsigned int readingBufferoffset = readingBufferPos; //Position where we start to
read from the original buffer
   unsigned short branchDepth = buffer[readingBufferoffset]; //Initial branch depth
   buffer[writtingBufferPos] = branchDepth;
   unsigned int index = 1;
   unsigned short branchCreated = 0;// Number of branch created
   unsigned int writtingIndex = 1; //WritingIndex in the new branch
   unsigned int writtingOffset = writtingBufferPos; // Where we start to write
   bool correctBranch = true; //Are the branches ok?
   while (index < Config::MAX_WIDTH && buffer[readingBufferoffset + index] != 0)
   { //Get the rule from the symbol
     vector<unsigned short>translate=_dic.Replace(buffer[readingBufferoffset+ index]);
      unsigned short depth = branchDepth;
      for (int i = 0; i < translate.size(); i++)
      {
       if (translate[i] == 1)//New Branch
       {
              if (i != 0)
              {
                     if (GenerateNewBranch(buffer, &branchCreated, &writtingIndex,
                          &depth, writtingOffset))
                            correctBranch = true;
                     else
                            return 0;//Could not create the new branch
              }
              else
                     depth += 1;
              correctBranch = true;
       }
       else if (translate[i] == 2)//Close branch
       {
              depth-=1;
              correctBranch = false;
       }
       else//Populate the branch
       {
              if (!correctBranch)
              {
                     if (GenerateNewBranch(buffer, &branchCreated, &writtingIndex,
                           &depth, writtingOffset))
                            correctBranch = true;
                     else
                            return 0;
              }
              if (writtingIndex < Config::MAX_WIDTH)
                  buffer[branchCreated * Config::MAX_WIDTH + writtingIndex +
                      writtingOffset] = translate[i];
              writtingIndex++;
       }
      }
      index++;
   }
   if ((branchCreated + 1) * Config::MAX_WIDTH + writtingOffset >=
       Config::MAX_BRANCHES * Config::MAX_WIDTH * Config::MAX_GENERATIONS)
        throw("Error FALTA MEMORIA");

   return (branchCreated + 1) * Config::MAX_WIDTH + writtingOffset;
}
```

Document 2. Codi per preparar els buffers d'un compute shader

Document referenciat en l'apartat *9.3.2 Càlcul de transformacions a nivell de branca*.

Estructura dels buffers dins de la classe ComputeShader, està compost pels següents buffers:

- Dades dels arbres *input buffer*.
- Emmagatzematge dels transformacions associades a les operacions bàsiques del L-System.
- El buffer de sortida del compute shader 1, on es guardaran les transformacions a nivell de branca.
- El buffer de sortida del compute shader 2, que és guarden de forma compacte els transformacions a nivell d'arbre.
- Els pares de cada branca per poder reconstruir la jerarquia de l'arbre.
- El nombre de troncs per arbre.

Tot tractar-se d'una tupla, hi ha els constructors definits ja que permetem pre-processar les dades inicials provinents del mòdul del L-System per adaptar-les als nostres buffers.

```
/**
* Struct for shader buffers.
* Stores all the buffer data. In only can hold two buffers, (input and output).
*/
struct buffers
{
   vector< GLushort > inputBuffer; //Tree Buffer
   vector < glm::mat4 > translateBuffer; // Buffer with transformations
   vector < glm::mat4 > computedBuffer; // OutputBuffer for C.Shader 1
   vector < glm::mat4 > compactComputedBuffer;//OutputBuffer for C.Shader 2
   vector < short > parentBufferIndices; //Buffer for parents
   vector < short > treeChunksCount;//Indice Buffer
   unsigned short num_branches;
   const unsigned int max_items_branch = 16;
   buffers()
   {

   }

   buffers& operator=(const buffers& other) // copy assignment
   {
      inputBuffer = other.inputBuffer;
      translateBuffer = other.translateBuffer;
      computedBuffer = other.computedBuffer;
      compactComputedBuffer = other.compactComputedBuffer;
      num_branches = other.num_branches;
      parentBufferIndices = other.parentBufferIndices;

      return *this;
   }
   /**
   * Initializes
   * @param <unsigned> input_size: size of the input vector.
   * @param <unsigned> input_init: value of the elements of input vector.
   * @param <unsigned> output_size: size of the output vector.
   * @param <unsigned> output_init: value of the elements of output vector.
   */
   buffers(unsigned short * branches, unsigned int branches_start, unsigned int
branches_end, glm::mat4 * translate, unsigned int translate_size)
   {

      for (unsigned int i = branches_start; i < branches_end; i++)
        inputBuffer.push_back(branches[i]);


      for (unsigned int i = 0; i < translate_size; i++)
        translateBuffer.push_back(translate[i]);

      num_branches = unsigned short(inputBuffer.size() / Config::MAX_WIDTH);

      computedBuffer.resize(num_branches * max_items_branch,glm::mat4());
      compactComputedBuffer.resize(num_branches * max_items_branch - 2,glm::mat4());
      parentBufferIndices.resize(num_branches, -1);

   }
};
```

Pel que fa la preparació dels buffers per ser enviats a la GPU, el procediment és igual per a tots els buffers:

- Generar el buffer amb la comanda glGenBuffers();
- Activar el buffer amb la comadna glBindBuffer();
- Entrar les dades amb glBufferdata();

```
void ComputeShader::BufferSetUpBrancheCompute()
{
   //Generate buffers
      // Input Buffer
   cerr << "Input Buffer generating, binding and filling ..." << endl;

   glGenBuffers(1, &_information.inputBuffer);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, _information.inputBuffer);
   glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(GLushort)* _buff.inputBuffer.size(),
&_buff.inputBuffer[0], GL_DYNAMIC_DRAW);

   glGenBuffers(1, &_information.translateBuffer);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, _information.translateBuffer);
   glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(glm::mat4)*
_buff.translateBuffer.size(), &_buff.translateBuffer[0], GL_DYNAMIC_DRAW);

   //Output Buffer
   cerr << "Output Buffer generating, binding and filling ..." << endl;
   glGenBuffers(1, &_information.computedBuffer);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, _information.computedBuffer);
   glBufferData(GL_SHADER_STORAGE_BUFFER, sizeof(glm::mat4)*
_buff.computedBuffer.size(), NULL, GL_STREAM_DRAW); //Not sure about GL_STREAM_DRAW,
more info at: https://www.khronos.org/registry/OpenGL-
Refpages/gl4/html/glBufferData.xhtml

   glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0); // <--Not sure about this one, maybe a
mistake

}
```

Document 3. Operació per executar compute Shader

Document referenciat en l'apartat *9.3.2 Càlcul de transformacions a nivell de branca*.

El codi per executar un compute shader consta de 3 parts:

1. Indicar quins buffers s'utilitzaran amb la comanda *glBindBufferBase()*.
2. Indicar quin programa es vol executar amb la comanda *glUseProgram()*.
3. Executar el compute shader amb la comanda *glDispatchCompute(x,y,z)*, on x,y,z és la mida 3D del compute shdaer, és adir, el nombre de Workgroups que s'executaran per cada eix.

```
void ComputeShader::ComputeBranch()
{
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, _information.inputBuffer);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, _information.translateBuffer);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, _information.computedBuffer);

   glUseProgram(_information.programID);

   glDispatchCompute(_buff.num_branches, 1, 1);


   glBindBuffer(GL_SHADER_STORAGE_BUFFER, _information.computedBuffer);
   glGetBufferSubData(GL_SHADER_STORAGE_BUFFER, 0,
sizeof(glm::mat4)*_buff.computedBuffer.size(), &_buff.computedBuffer[0]);
see https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetBufferSubData.xhtml

}
```