Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques

Universitat de Barcelona

Implementació CPU/GPU d'una aplicació per a la deformació de models 3D amb Mean Value Coordinates

Albert Busqué Plaza

Director: Lluis Garrido Ostermann

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi

Barcelona, 17 de Gener del 2013

Índex

Capítol 1

Introducció

Avui en dia, l'animació s'ha convertit en un aspecte molt important en molts sectors empresarials així com també a nivell d'entreteniment. Qui no ha vist una pel·lícula d'animació per ordinador o ha jugat a algun videojoc? És ben cert que cada vegada estem més acostumats a veure gràfics més detallats i realistes i que, si mirem els de fa una dècada, ens n'adonarem de la rapidesa amb que tot això ha evolucionat. Això fa que, inconscientment, nosaltres mateixos siguem més exigents i crítics quan ens posem davant d'una escena tridimensional.

Si ens ho mirem des d'un punt de vista tècnic, aquesta ambició per aconseguir escenes tridimensionals més perfectes pot esdevenir força cara: més realisme implica més informació. És evident, doncs, que cada vegada tractem amb escenes que poden contenir milers o fins i tot milions de punts i polígons, normalment triangles, que s'han de manipular amb la major rapidesa possible. Això planteja un seguit de qüestions: com ho fem per animar un objecte? Ho hem de fer punt per punt? I encara més important, com ho podem fer de forma ràpida i eficient?

Molts investigadors de l'àmbit dels gràfics per ordinador estan treballant en aquest problema. Fins ara, s'ha aconseguit trobar un grapat de tècniques i algorismes [1, 2, 3, 4]. Aplicacions d'edició gràfica tant importants com

Autodesk Maya, Autodesk 3ds Max o Blender incorporen aquestes tècniques per donar als usuaris la possibilitat de modelar i animar objectes. Destacar també, com empreses tant importants com Pixar Animation Studios fan recerca en aquest àmbit (consultar article "Harmonic Coordinates for Character Articulation"[3]) i utilitzen aquestes tècniques per produir les seves pel·lícules.

Aquest treball s'emmarca en el camp de gràfics per ordinador i més concretament l'àmbit de la mesh deformation, computational geometry i object modeling. Així mateix, també es pot incloure dins del camp de la programació paral·lela, ja que es fa una incursió en aquest paradigma de programació per aprofitar les avantatges que suposa fer els diferents càlculs a la targeta gràfica. El motiu pel qual s'utilitza paral·lelisme és degut al gran volum de càlculs que es realitzen a l'aplicació per tal d'aconseguir l'efecte de deformació d'un model 3D. Per això, esperem, amb la introducció d'aquest paradigma, poder accelerar l'execució de l'aplicació i millorar-ne el rendiment paral·lelitzant les parts que tenen una càrrega de càlculs intensa.

Aquest treball té l'objectiu d'estudiar què hi ha a baix nivell en relació a l'animació per ordinador, és a dir, la deformació d'un model. Però, com s'aconsegueix aquesta deformació? De forma abstracta, animar un model o personatge no és res més que fer un conjunt de deformacions sobre un model tridimensional de forma incremental de tal manera que aquest aconsegueixi un efecte de moviment i sembli que tingui vida pròpia. En particular, s'hi trobarà l'estudi i implementació de l'algorisme mean value coordinates el qual permet aconseguir la deformació d'una malla triangular tancada, tant a CPU com a GPU.

1.1 Resum en anglès

In this work we will develop an application to deform 3D models. To perform deformation, we implement mean value coordinates for closed triangular

meshes algorithm. These models are sorrounded by a cage. To deform them, we stretch a cage's point interactively by using mouse. For displaying results and interact with model we will use free open source VTK library. In addition, we will add parallelism in GPU to certain parts of the code to try to improve performance using OpenCL framework.

Capítol 2

Objectius

L'objectiu principal d'aquest treball és desenvolupar una aplicació per deformar models tridimensionals on aquesta deformació és feta per l'usuari de manera interactiva amb l'ajut del ratolí. En particular, el model que es vulgui deformar s'envoltarà amb una "caixa" on l'usuari, seleccionant un vèrtex d'aquesta amb el ratolí, l'estirarà convenientment alhora que el model que conté s'anirà deformant d'acord amb la caixa. Per tant, aconseguirem la deformació del model 3D de forma indirecta modificant la caixa que l'envolta.

A més a més, es vol incloure paral·lelisme per augmentar el rendiment de l'aplicació. De forma resumida, tenim els objectius següents:

1. Deformar un model tridimensional.
2. Utilitzar programació paral·lela.

Per assolir aquests objectius, s'ha implementat una aplicació que té quatre parts bén diferenciades:

1. Lectura d'un fitxer contenidor d'un model tridimensional.
2. Implementació del mètode matemàtic que permeti aconseguir la deformació d'una malla triangular.

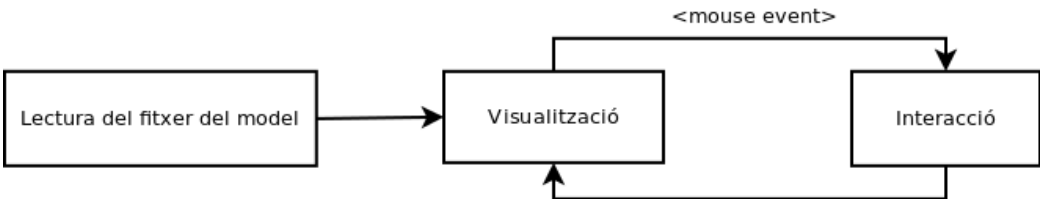


Figura 2.1: Funcionament abstracte de l'aplicació

3. Visualització i interacció de/amb el model tridimensional.

4. Afegir paral·lelisme a l'aplicació.

La idea abstracta i general de com ha de funcionar l'aplicació (veure figura 2.1) és la següent: en primer lloc, l'aplicació ha de llegir els punts i triangles del model d'un fitxer de text i visualitzar el model per pantalla; després entrarà en un bucle infinit que consisteix en: capturar la interacció de l'usuari amb el ratolí i visualitzar la deformació.

Capítol 3

Context

En aquest capítol s'explicarà quines tecnologies s'han escollit per desenvolupar l'aplicació així com l'entorn d'execució. Per explicar-ho, es seguirà els objectius descrits al capítol 2 detallant en cadascun les tecnologies utilitzades.

En primer lloc, dir que totes les eines utilitzades en el desenvolupament són gratuïtes i de codi obert. S'ha utilitzat el sistema operatiu Linux per desenvolupar, executar i testejar i el llenguatge de programació C/C++ per escriure l'aplicació de forma més eficient.

3.1 Algorisme per deformació de malles triangulars

Com ja s'ha comentat a la introducció, el mètode matemàtic que permet aconseguir la deformació d'una malla triangular és el "Mean Value Coordinates for Closed Triangular Meshes"[1] de Tao Ju, Scott Schaefer, Joe Warren fet a la Rice University que s'explicarà al capítol 4.

3.2 VTK

Per llegir el fitxer del model, visualitzar-lo i interactuar-hi s'ha utilitzat la llibreria VTK [7].

The Visualization Toolkit (VTK) és un sistema de programari lliure, de codi obert i multi-plataforma (funciona en Linux, Windows, Mac i Unix) desenvolupat per Kitware per a gràfics 3D per ordinador, processament d'imatges i visualització. VTK consisteix en una llibreria de classes C++ i diverses capes d'interfície interpretades com Tcl / Tk, Java i Python. Proporciona una API de més alt nivell respecte OpenGL que simplifica els resultats de la renderització i visualització. Aquesta API consisteix amb un conjunt d'objectes tals com càmeres, llums i actors que són continguts dins dels renders que apareixen a una finestra (veure figura 3.1). VTK té un marc de visualització d'informació àmplia, disposa d'un conjunt de widgets d'interacció 3D, suporta el processament paral·lel, i s'integra amb diverses bases de dades en eines GUI com Qt i Tk.

Com que la llibreria és orientada a objectes, tot el que és veu a la figura 3.1 són objectes: renders, finestres, actors, polígons... Una escena es composa bàsicament d'una finestra, el window render d'aquesta, un/s actor/s (objectes que es visualitzen a la pantalla), els seus mappers (l'objecte que fa el mapeig entre les dades de visualització i el motor gràfic) i un estil d'interacció. Les dades de visualització poden ser imatges, graelles, punts no estructurats, polígons... (veure figura 3.2).

3.3 OpenCL

Per afegir paral·lelisme s'ha emprat l'entorn de programació estàndard i gratuït OpenCL (Open Computing Language) [5]. Aquest ens permetrà paral·lelitzar les parts de l'aplicació que realitzen més càrrega de càlculs amb l'objectiu de millorar el temps d'execució.

OpenCL és un entorn estàndard de programació que s'executa en plataformes

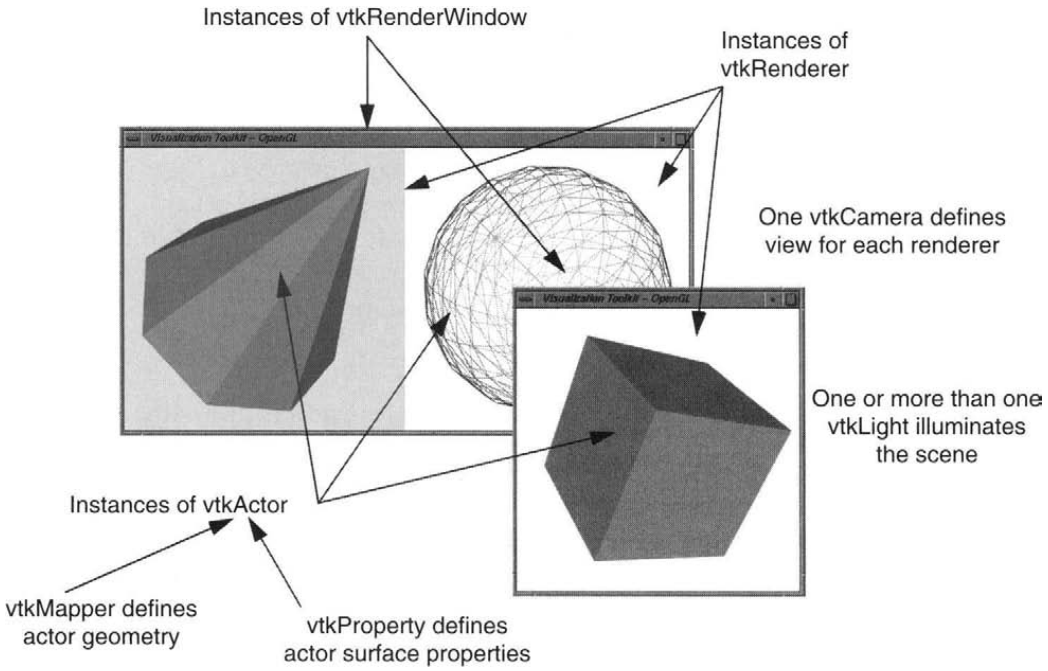


Figura 3.1: Model de finestres de l'VTK

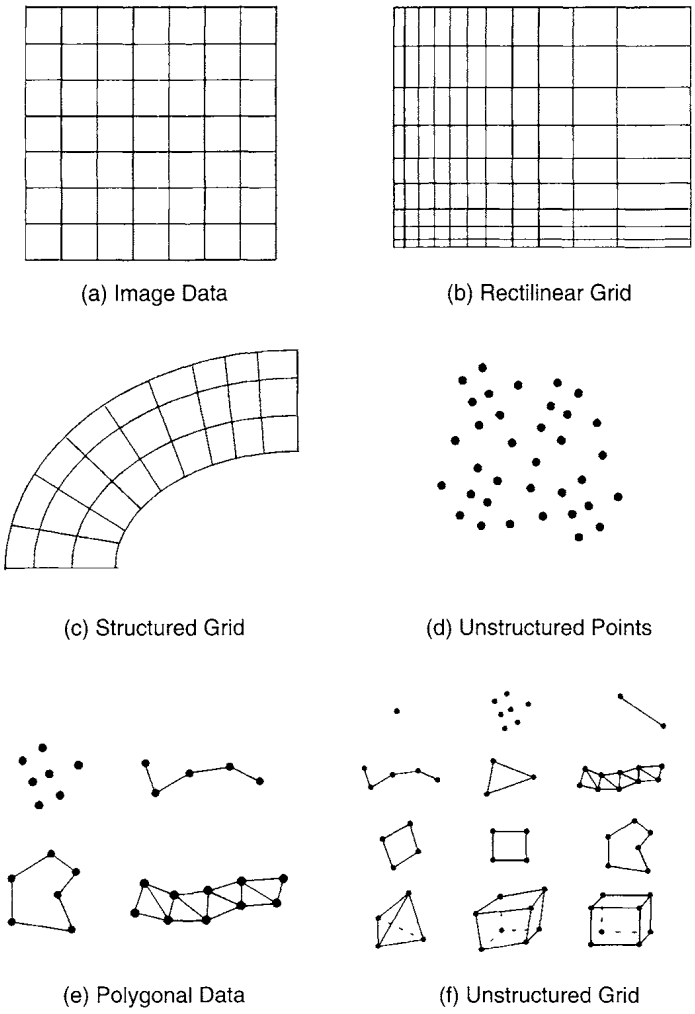


Figura 3.2: Dades de visualització de l'VTK

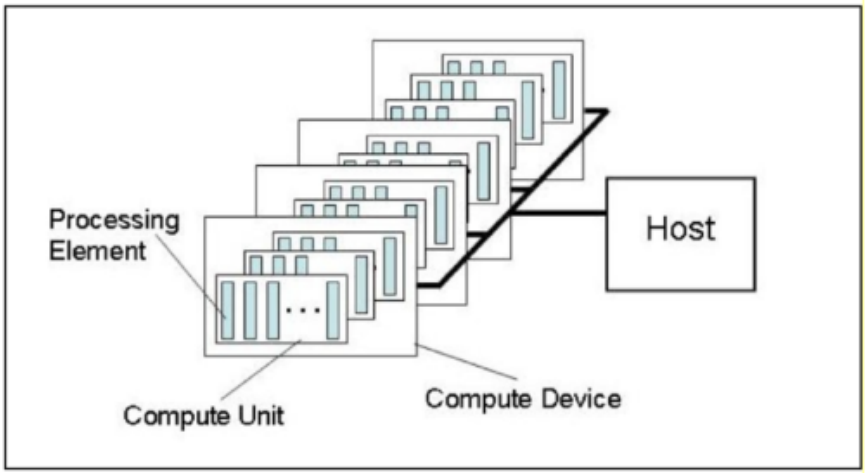


Figura 3.3: Model de la plataforma OpenCL

heterogènies, com ara unitats centrals de processament (CPU), unitats de processament gràfic (GPU), DSPs i altres processadors. Inclou un llenguatge (basat en C99) per escriure els kernels (funcions que s'executen en els dispositius OpenCL), a més d'una interfície de programació d'aplicacions (API) que s'utilitza per definir i controlar les plataformes.

OpenCL ofereix la computació paral·lela utilitzant paral·lelisme basat en tasques i basada en dades. És un estàndard obert mantingut pel consorci tecnològic sense ànim de Khronos Group i ha estat adoptat per Intel, Advanced Micro Devices, Nvidia, i ARM Holdings.

Com podem observar a la figura 3.3, el model d'OpenCL consisteix en un dispositiu anomenat "host" i en múltiples dispositius anomenats "compute devices o devices". El host és generalment la CPU i és l'encarregat de transferir les dades als devices i rebre els resultats d'aquests. Per altra banda, els devices són normalment dispositius GPU i són els encarregats de processar les dades en paral·lel gràcies als múltiples "compute unit" que tenen. Cada compute unit executa una instància d'un kernel.

Capítol 4

Mean value coordinates

És una tècnica descoberta per Tao Ju, Scott Schaefer, Joe Warren a la Rice University que s'empra en els camps de construcció volumètrica de textures i deformació de superfícies.

Suposem que tenim un punt $p = (x_0, y_0, z_0)$ en el sistema cartesià format pels vectors:

$$\vec{e_1} = (1, 0, 0)$$

$$\vec{e_2} = (0, 1, 0)$$

$$\vec{e_3} = (0, 0, 1)$$

observar que:

$$p = x_0\,\vec{e_1} + y_0\,\vec{e_2} + z_0\,\vec{e_3}$$

Direm que x_0, y_0, z_0 són les coordenades de p en el sistema e_1, e_2, e_3.

Suposem un conjunt de n punts $v = \{v_0 \dots v_n\}$ tal com la figura 4.1. La idea és trobar unes coordenades w de dimensió n tals que:

$$p = \sum_{i=1}^{N} w_i v_i \tag{4.1}$$

on $w_1, w_2, \dots, w_n$ són les coordenades de p en el sistema $v_1, v_2, \dots, v_n$ on

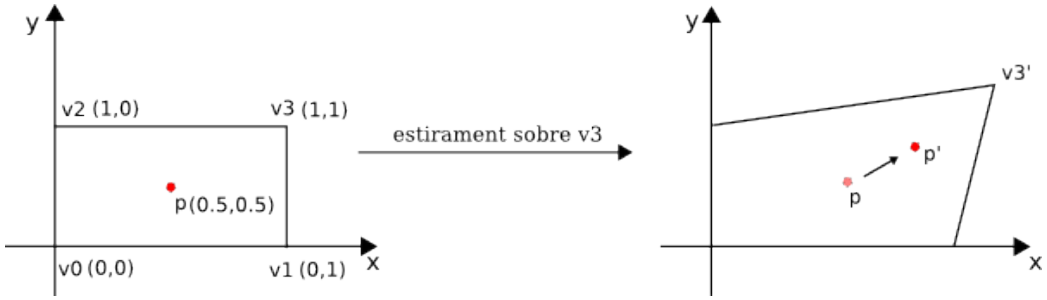


Figura 4.1: Esquerra: el punt p es descriu com a combinació lineal dels punts de la caixa v1, v2, v3 i v4 (veure equació 4.1). Dreta: en moure el punt v3 de la caixa, el punt p' es pot recuperar fent servir l'equació 4.2.

$v_1, \ldots, v_n$ són punts no necessàriament linealment independents.

Si les coordenades de qualsevol punt $v_i \in v$ es modifiquen, es recupera el punt p deformat p' amb:

$$p^{'} = \sum_{i=1}^{N} w_i v_i^{'} \tag{4.2}$$

En altres paraules, les coordenades w (a partir d'ara coordenades afins) són independents al moviment dels punts v de manera que si nosaltres movem qualsevol dels punt v_i qualsevol nombre de vegades on $v_i \in \vec{v}$, el punt p de la figura es desplaçarà amb concordança amb el moviment fet per v_i amb la avantatge que no caldrà calcular les coordenades afins cada vegada que es mogui v_i.

Des de la perspectiva de la nostra aplicació, els punts v fan la funció d'una caixa que envolta el punt p i que ens serveix per modificar aquest modificant únicament la caixa. Si extrapolem aquest raonament a un escenari més complex podem pensar que, si tenim un model tridimensional amb n punts, podem envoltar tots aquests els punts de la mateixa manera que ho fèiem en el cas de la figura 4.1; en altres paraules, envoltem el model 3D amb una caixa. Per tant, podem intuir que tots els punts del model es mouran en funció del desplaçament que s'hagi produït en qualsevol punt de la caixa

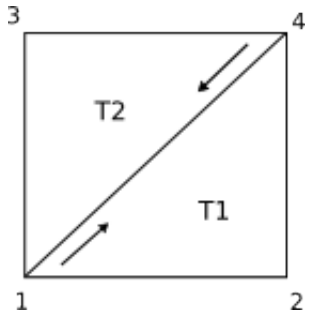
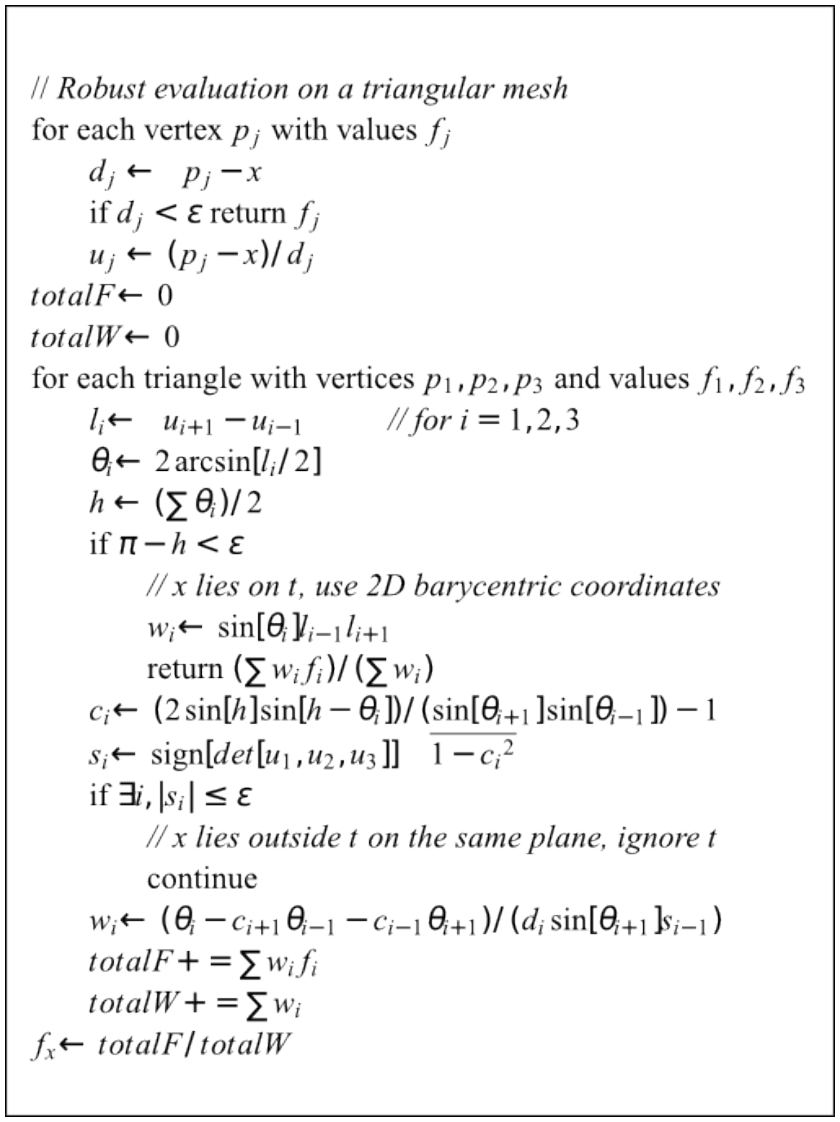


Figura 4.2: Exemple d'una correcta orientació dels triangles

aconseguint un efecte de deformació en el model.

En el cas de les mean value coordinates, aquesta caixa ha d'estar formada per una superfície tancada formada per triangles. Cal ressaltar que els vèrtex de la caixa han d'estar orientats de forma positiva de tal manera que quan l'algorisme recorri un determinat triangle i aquest tingui una aresta comuna amb un triangle veí, l'aresta haurà d'ésser recorreguda en sentit oposat a la iteració del triangle veí. Per exemple, si observem la figura 4.2, veiem una caixa de forma quadrada partida en dos triangles T_1 i T_2 on l'aresta comuna és la $\{1, 4\}$. Una orientació correcte per aquest exemple és descriure $T_1 = \{1, 4, 2\}$ i $T_2 = \{4, 1, 3\}$. D'aquesta manera, l'aresta comuna és descrita en ambdós sentits (un per cada triangle) tal com indiquen les fletxes.

Com hem comentat abans, necessitem saber les coordenades afins de cada punt del model 3D que ens expressen un punt del model en el sistema format pels punts de la caixa. Per fer aquest càlcul, es segueix l'algorisme de la figura 4.3 que es proposa a l'article [1].



// *Robust evaluation on a triangular mesh*
for each vertex p_j with values f_j
    $d_j \leftarrow p_j - x$
    if $d_j < \varepsilon$ return f_j
    $u_j \leftarrow (p_j - x)/d_j$
$totalF \leftarrow 0$
$totalW \leftarrow 0$
for each triangle with vertices p_1, p_2, p_3 and values f_1, f_2, f_3
    $l_i \leftarrow u_{i+1} - u_{i-1}$ // *for* $i = 1,2,3$
    $\theta_i \leftarrow 2\arcsin[l_i/2]$
    $h \leftarrow (\sum \theta_i)/2$
    if $\pi - h < \varepsilon$
        // *x lies on t, use 2D barycentric coordinates*
        $w_i \leftarrow \sin[\theta_i]l_{i-1}l_{i+1}$
        return $(\sum w_i f_i)/(\sum w_i)$
    $c_i \leftarrow (2\sin[h]\sin[h-\theta_i])/(\sin[\theta_{i+1}]\sin[\theta_{i-1}]) - 1$
    $s_i \leftarrow \text{sign}[det[u_1, u_2, u_3]] \sqrt{1 - c_i^2}$
    if $\exists i, |s_i| \leq \varepsilon$
        // *x lies outside t on the same plane, ignore t*
        continue
    $w_i \leftarrow (\theta_i - c_{i+1}\theta_{i-1} - c_{i-1}\theta_{i+1})/(d_i \sin[\theta_{i+1}]s_{i-1})$
    $totalF += \sum w_i f_i$
    $totalW += \sum w_i$
$f_x \leftarrow totalF/totalW$

Figura 4.3: Algorisme mean value coordinates for closed triangular meshes [1].

Capítol 5

Implementació

En aquest capítol s'explicarà com s'ha implementat l'aplicació.

En primer lloc, comentar que s'ha desenvolupat en llenguatge C a excepció del mòdul de visualització que ha estat escrit en C++. Per la compilació s'utilitza l'eina gratuïta CMake (Cross Platform Make) que permet compilar codi per diverses plataformes.

A la figura 5.1 hi ha representat el diagrama de flux de l'aplicació. Com es pot observar, l'aplicació realitza cinc passos: càrrega del model i la caixa a memòria, càlcul coordenades afins (mean value coordinates), visualització per pantalla, actualització del punt de la caixa i càlcul dels nous punts del model 3D.

5.1 Càrrega del model i la caixa a memòria

És el primer pas que realitza l'aplicació. Abans d'explicar com s'ha implementat aquest pas, és necessari explicar com és un model.

Els models estan especificats en fitxers ply, obj, 3ds etc on cadascun té un format diferent. En tots ells s'hi troben els punts x, y, z (en el cas tridimensional) i els polígons, que en el nostre cas són triangles. Cada triangle està representat amb tres sencers que corresponen amb els índex dels punts

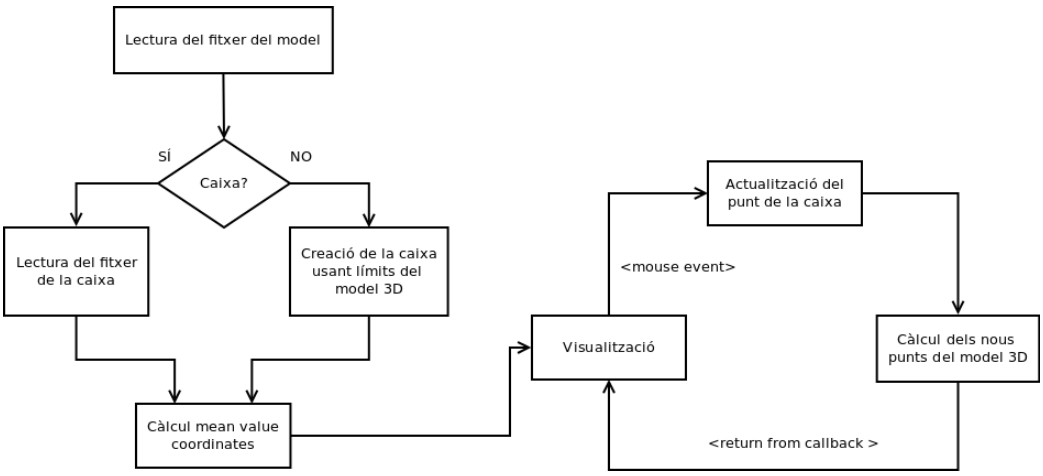


Figura 5.1: Diagrama de flux corresponent a l'aplicació.

que el formen; per exemple, si tenim p0, p1 i p2 i aquests tres punts formen un triangle, aquest es representaria com 0 1 2 (veure figura 5.2 per més claredat). La combinació dels punts i els triangles formen una malla (mesh en anglès).

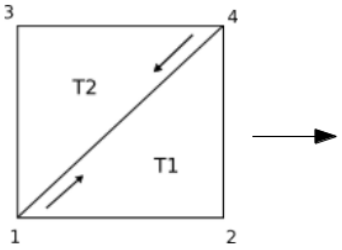
Bàsicament, carregar o llegir un model implica la creació de tres objectes de la llibreria VTK en l'ordre següent:

1. vtkPolyData: és l'objecte que conté els punts i polígons que formen el model.

2. vtkPolyDataMapper: és l'objecte encarregat de fer el mapeig entre les dades de visualització i el motor gràfic; és a dir, transforma els punts cartesians del model a píxels.

3. vtkActor: és l'objecte objecte que representa un actor dins d'una escena.

Quan es carrega un model des d'un fitxer, es llegeix seqüencialment tots els punts i triangles i es copien dins de l'objecte vtkPolydata. Afortunadament, si volem fer servir formats estàndard, no cal escriure la funció que faci això



## Punts		
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
## Triangles		
1	4	2
4	1	3

Figura 5.2: Exemple d'un fitxer d'un model

ja que la llibreria de visualització VTK incorpora un grup de classes que automatitzen aquest procés. En particular, la nostra aplicació utilitza les classes vtkPLYReader i vtkOBJReader per llegir fitxers amb extensions PLY i OBJ respectivament i automatitzar la creació dels objectes vtkPolyData i vtkPolyDataMapper.

A continuació s'explica més en detall la càrrega del model 3D i la caixa.

5.1.1 Càrrega del model 3D

Aquesta subsecció explicarà la caixa "Lectura del fitxer del model" de la figura 5.1.

En aquest pas, es llegeix el model 3D a partir del fitxer que l'usuari ha especificat com a entrada de l'aplicació. Tal com s'ha comentat a la secció anterior, aquesta lectura implica la creació de tres objectes (vtkPolyData, vtkPolyDataMapper, vtkActor) per tal que la llibreria de visualització pugui dibuixar-lo per pantalla. A més a més, tindrem carregats a memòria (dins d'un objecte vtkPolyData) els punts cartesians i els triangles.

5.1.2 Càrrega de la caixa

Aquesta subsecció explicarà la caixa "Lectura del fitxer de la caixa" i "Creació de la caixa usant límits del model 3D" de la figura 5.1.

A l'hora de carregar la caixa, tenim dues possibilitats en funció de què hagi especificat l'usuari:

1. Llegir-la des d'un fitxer de forma anàloga a la carrega del model 3D vist a l'apartat 5.1.1.

2. Crear-ne una per defecte, en forma d'ortoedre, a partir dels límits (x, y, z) del model 3D.

El primer cas serveix per a carregar una caixa especifica per un model 3D determinat. D'aquesta manera, s'aconsegueix una deformació més acurada ja que la caixa s'adapta perfectament al contorn del model.

El segon cas és útil quan no es té la caixa específica del model ja que habitualment els models no adjunten la caixa. D'aquesta manera podem deformar qualsevol model i eliminar la restricció de tenir una caixa específica. En contraposició, el fet d'utilitzar una caixa "genèrica" farà que la deformació sigui menys detallada que l'aconseguida en el primer cas.

5.2 Càlcul de les coordenades afins (mean value coordinates)

Aquesta subsecció explicarà la caixa "Càlcul mean value coordinates" de la figura 5.1.

Després de la càrrega dels models, ve el pas on es calculen les coordenades afins del model 3D. Tal com hem vist al capítol 4, aquestes es calculen respecte un punt i una caixa. Si suposem una model f amb n punts i una caixa c amb m punts, hi haurà n coordenades afins de dimensió m, és a dir,

Algorisme 5.1 Fragment que il·lustra el càlcul de les coordenades afins

```
FOR each point IN model
{
        compute_coordinates(cage, point, output);
}
```

tantes coordenades afins com punts del model de dimensió número de punts de la caixa.

Per fer el càlcul de les coordenades afins, s'itera seqüencialment sobre tots els punts del model 3D i s'executa l'algorisme sobre cadascun d'ells juntament amb la informació de la caixa (veure algorisme 5.1). En particular, la funció que implementa l'algorisme "Mean Value Coordinates for Closed Triangular Meshes" (veure figura 4.3) s'anomena "compute_coordinates".

5.3 Visualització

Aquesta subsecció explicarà la caixa "Visualització" de la figura 5.1.

Aquest pas és l'encarregat de dibuixar el model 3D i la caixa per pantalla. En particular, qui fa aquesta feina és la llibreria VTK que a la vegada, es recolza sobre la llibreria OpenGL.

A grans trets, els passos per iniciar la visualització estan representats a l'algorisme 5.2. L'aspecte més important pel que fa a la visualització es troba a la definició l'objecte interacció. Aquest objecte conté la manera com respondrà el sistema davant dels esdeveniments que produeixi l'usuari amb el ratolí. Per tant, és aquí on s'implementa què s'ha de fer quan l'usuari clica un punt de la caixa i el desplaça.

Com en tota interfície gràfica, cada botó té assignat una funció (també anomenada callback) que s'executa cada cop que es produeix un esdeveniment (pitjar un botó, moure el ratolí, moure la rodeta...). En particular, hem redefinit les funcions que corresponen als esdeveniments "clicar el botó del

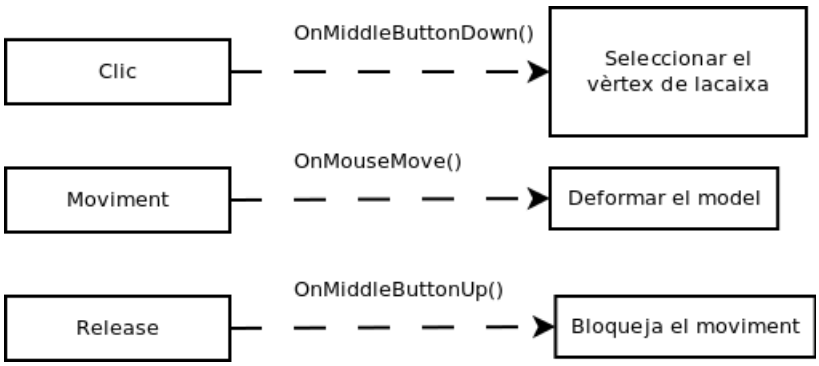


Figura 5.3: Relació entre events , callbacks i accions

Algorisme 5.2 Passos en l'arrencada de la visualització

```
CREATE Renderer, Render Window, Interactor
CREATE Point Picker
SET Interactor_Style ## defines our type of interaction
ADD actors to the scene
START render and interaction ## enters an infinite loop
```

mig", "deixar de clicar el botó del mig", i "moure el ratolí" per estendre'n la funcionalitat (veure figura 5.3). Al primer és on es captura el vèrtex de la caixa que l'usuari a clicat; al segon, és on es calcula i visualitza la deformació del model; finalment, al tercer és on es captura que el botó deixa d'ésser pitjat i fa que OnMouseMove no faci res.

5.4 Actualització del punt de la caixa

Aquesta subsecció explicarà la caixa "Actualització del punt de la caixa" de la figura 5.1.

Tal com s'ha dit a la secció 5.3, l'usuari clica el vèrtex de la caixa que vol estirar per tal de deformar el model. Per tal d'actualitzar la caixa, necessitem saber quin vèrtex vol moure's i la seva posició final.

El vèrtex (més aviat el seu identificador) es captura en el moment en que

Algorisme 5.3 Càlcul d'un punt a partir de les coordenades afins i una caixa.

```
PER k=0; k<cada punt del model; k++
        p_k <- inicialitzem p_k a 0
        PER i=0; i<cada punt de la caixa; i++
                pc_i <- punt i-èssim de la caixa
                p_k = mvcoords[k][i] * pc_i
        // Update k-èssim punt
        k-èssim point <- p_k;
```

l'usuari clica sobre un vèrtex generant un esdeveniment que és recollit per la funció OnMiddleButtonDown. Aquesta captura el píxel clicat i el tradueix en un identificador de vèrtex de la caixa (en cas que s'hagi clicat correctament). És en el moment que es mou el ratolí, que es genera un altre esdeveniment recollit per la funció OnMouseMove i es captura la posició x,y,z final del vèrtex (que correspon amb la posició del ratolí). Finalment, s'actualitza el punt de la caixa amb l'identificador i la nova posició.

5.5 Càlcul dels nous punts del model

Aquesta subsecció explicarà la caixa "Càlcul dels nous punts del model 3D" de la figura 5.1.

Després d'actualitzar el vèrtex mogut de la caixa, s'actualitzen tots els punts del model 3D a partir de les coordenades afins i la "nova" caixa. El procediment és el següent: s'itera sobre tots els punts del model i per cada punt d'aquest s'itera sobre cada punt de la caixa. Dins d'aquest segon bucle, es fa el càlcul de la recuperació del punt multiplicant la coordenada afí i-èssima del punt k amb el punt i-èssim de la caixa (veure algorisme 5.3). Noteu que el punt es composa de tres dimensions que no estan explícitament representades en l'algorisme 5.3 en el moment de fer la multiplicació.

Capítol 6

Paral·lelització

En aquest capítol tractarem quines parts i com s'ha paral·lelitzat l'aplicació mitjançant l'entorn de programació OpenCL.

En els darrers anys, ha sorgit un nou paradigma de programació basat en el paral·lelisme degut a la necessitat de buscar noves maneres d'accelerar l'execució dels programes en detriment d'augmentar la velocitat dels microprocessadors. El paral·lelisme es basa en executar més d'una instrucció al mateix temps (i.e. en paral·lel); això implica entre altres coses, la possibilitat de fer més d'una operació aritmètica en un mateix instant de temps. Sobre el paper, si una màquina és capaç de fer n càlculs en paral·lel, hauria d'obtenir un guany computacional per valor de factor n.

La unitat bàsica d'execució de l'OpenCL s'anoma kernel. Un kernel és un conjunt d'instruccions que s'executen en paral·lel. Per entendre-ho millor, podem pensar en un relació de semblança entre un fil d'execució (thread) d'una CPU i un kernel i que, quan es paral·lelitza alguna cosa, estem creant molts fils d'execució que tenen un comportament idèntic però actuen sobre posicions diferents dins d'un mateix espai de dades.

Alhora de paral·lelitzar una aplicació, es tenen dues possibilitats:

1. Sobre la CPU (aquesta ha d'ésser multicore), utilitzant múltiples fils d'execució. Aquesta opció aprofita els diversos nuclis que disposen les

CPU modernes.

2. Sobre la GPU, utilitzant l'alta capacitat de càlcul paral·lel que tenen aquests dispositius. Aquesta opció s'anomena GPGPU (General-purpose graphics processing unit).

Afortunadament, l'OpenCL és un entorn capaç de paral·lelitzar en ambdós dispositius de manera transparent pel desenvolupador en funció de quin dispositiu hi ha disponible. A més a més, proporciona un nivell d'abstracció envers el maquinari de tal manera que una mateixa implementació és vàlida per a múltiples dispositius.

Com s'ha comentat en els objectius (veure capítol 2), hi ha parts de l'aplicació que s'han paral·lelitzat per tal d'augmentar-ne el rendiment. A continuació s'identifiquen i s'expliquen les parts en qüestió.

6.1 Càlcul de les coordenades afins

A diferència del cas seqüencial vist a la secció 5.2, no s'utilitza cap iteració per calcular les coordenades afins punt per punt; la idea és calcular-les de cop, és a dir, en paral·lel. Això farà que puguem calcular-les utilitzant menys temps que si ho féssim seqüencialment.

Intuïtivament, la idea és executar alhora múltiples instàncies de la funció que implementa l'algorisme de la figura 4.3. Aquesat funció, tal com es pot veure a la secció 5.2, s'anomena compute_coordinates. En particular, el que s'ha fet és modificar compute_coordinates per tal de convertir-la en un kernel que OpenCL serà capaç d'executar a la targeta gràfica en paral·lel.

Aquest, té molt poques diferències a nivell de programació amb la funció compute_coordinates utilitzada en el cas seqüencial. El més destacat que s'ha hagut de modificar ha sigut els paràmetres d'entrada i els index d'accés a les posicions de memòria com a conseqüència d'una reorganització de les dades. Com que OpenCL treballa únicament amb punters d'una dimensió,

Algorisme 6.1 Capçalera del kernel compute_coordinates

```
__kernel void compute_coordinates (
                        unsigned int cage_npts,
                        __global float *cage_points,
                        unsigned int nfaces,
                        __global unsigned int *faces,
                        __global float *x,
                        __global float *w,
                        __global float *dist,
                        __global float *u)
```

s'ha hagut de re-estructurar totes les dades per tal d'empaquetar-les dins d'arrays unidimensionals. D'aquesta manera, els paràmetres corresponents als punts cartesians, els triangles i els vectors d'escriptura que utilitza l'algorisme són tires de flotants (veure algorisme 6.1).

6.2 Càlcul dels nous punts del model

La implementació paral·lela del càlcul dels nous punts del model està basat en l'algorisme 5.3. A diferència del cas seqüencial vist a la secció 5.5, la idea és executar al mateix temps múltiples instàncies de l'algorisme seqüencial. Per això, s'ha implementat el kernel representat a l'algorisme 6.2. Cada instància d'aquest kernel calcularà el nou punt i-èssim del model 3D.

A grans trets, es realitzen tres passos pel que fa al càlcul (veure figura 6.1):

1. Transferència de caixa cap a la GPU.
2. Càlcul dels nous punts del model 3D de forma para·lela a la GPU.
3. Transferència dels nous punts de la GPU cap a la CPU.

Cal recordar que aquests passos s'executen a cada moviment del ratolí mentre es mou un vèrtex determinat de la caixa. En el primer pas, es fa una

Algorisme 6.2 Kernel de la recuperació del punt

```
__kernel void recover_point (
                        __global float *mvcoords,
                        __global float *cage,
                        unsigned int cage_npts,
                        __global float *out)
{
        float out_x = 0.0f, out_y = 0.0f, out_z = 0.0f;
        unsigned int i;

        // Get global index in mvcoords array
        unsigned int idx = get_global_id(0);
        unsigned int k = idx*cage_npts;

        // Calculate recovered point p
        for (i=0; i<cage_npts; i++)
        {
                out_x += mvcoords[k+i] * cage[i*3+0];
                out_y += mvcoords[k+i] * cage[i*3+1];
                out_z += mvcoords[k+i] * cage[i*3+2];
        }

        // Store point p in out
        out[idx*3+0] = out_x;
        out[idx*3+1] = out_y;
        out[idx*3+2] = out_z;
}
```

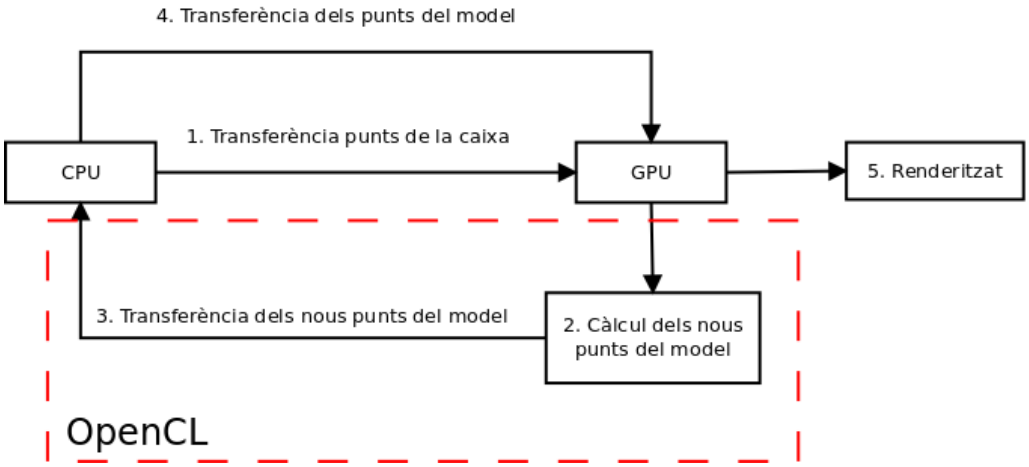


Figura 6.1: Flux en el càlcul dels nous punts del model amb paral·lelització

transferència dels punts de la caixa cap a la GPU. En el segon, es realitza el càlcul dels nous punts amb OpenCL de forma paral·lela. En el tercer, es transfereixen els punts calculats al punt dos cap a la CPU i es sobreescriuen els vells a memòria principal. A més a més, aquests nous punts són transferits de nou a la GPU per tal que l'OpenGL pugui renderitzar-los.

Capítol 7

Resultats

En aquest capítol s'exposaran unes quantes imatges corresponents als models deformats per donar una idea del resultat obtingut. A més a més, s'hi trobarà una comparativa entre l'execució "tradicional" i la paral·lela per analitzar-ne el rendiment.

7.1 Entorn de proves

El nostre entorn de proves és el següent:

- Sistema Operatiu: Ubuntu 12.04.
- CPU: Intel Core™ i3-2100 Processor (3M Cache, 3.10 GHz).
- GPU: AMD Radeon™ HD 7750.
- 4 GB DDR3-1333.

7.2 Dataset

A continuació es presenten els models que s'han fet servir per realitzar les proves.

- Happy Buddha
  - Origen: Stanford University Computer Graphics Laboratory.
  - Dimensions : 543,652 vèrtex, 1,087,716 triangles.
  - Extret de The Stanford 3D Scanning Repository [8].
  - Caixa: per defecte, 8 vèrtex, 12 triangles.
- Stanford Bunny
  - Origen: Stanford University Computer Graphics Laboratory.
  - Size of reconstruction: 35947 vèrtex, 69451 triangles.
  - Extret de The Stanford 3D Scanning Repository [8].
  - Caixa: per defecte, 8 vèrtex, 12 triangles.
- Camell
  - Origen: Universitat de Girona.
  - Dimensions: 9770 vèrtex, 19536 triangles.
  - Caixa: 36 vèrtex, 68 triangles.

7.3 Deformació

A la figura 7.1 podem veure-hi tres deformacions fetes al model del camell utilitzant una caixa específica. A la primera imatge hi ha el model original i a les següents el model deformat en zones diferents. De forma anàloga, a les figures 7.2 i 7.3 podem veure-hi el model Standford Bunny i Happy Buddha respectivament deformats amb una caixa per defecte en forma d'ortoedre de 8 punts i 12 triangles.

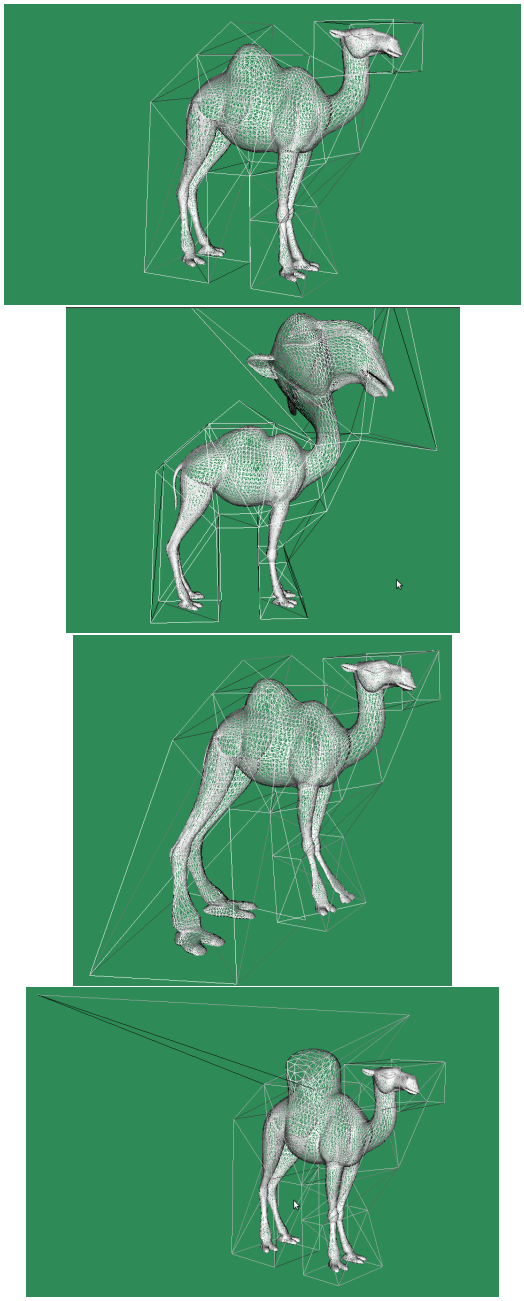


Figura 7.1: Resultats en la deformació del model Camell.

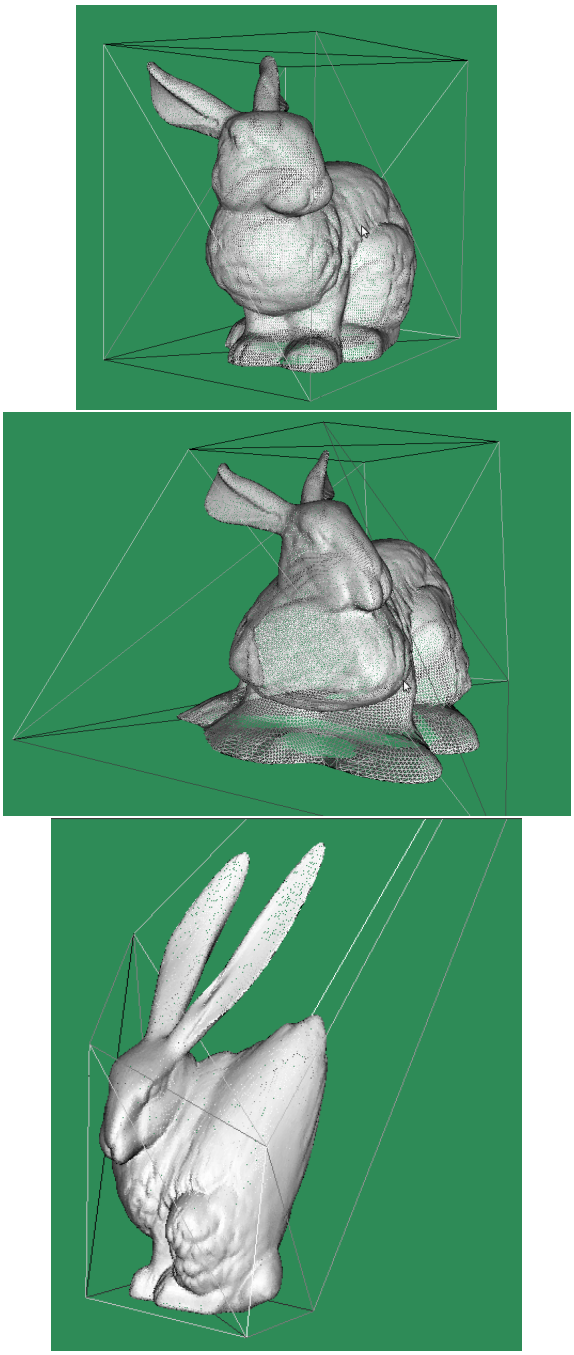


Figura 7.2: Standford Bunny deformat.

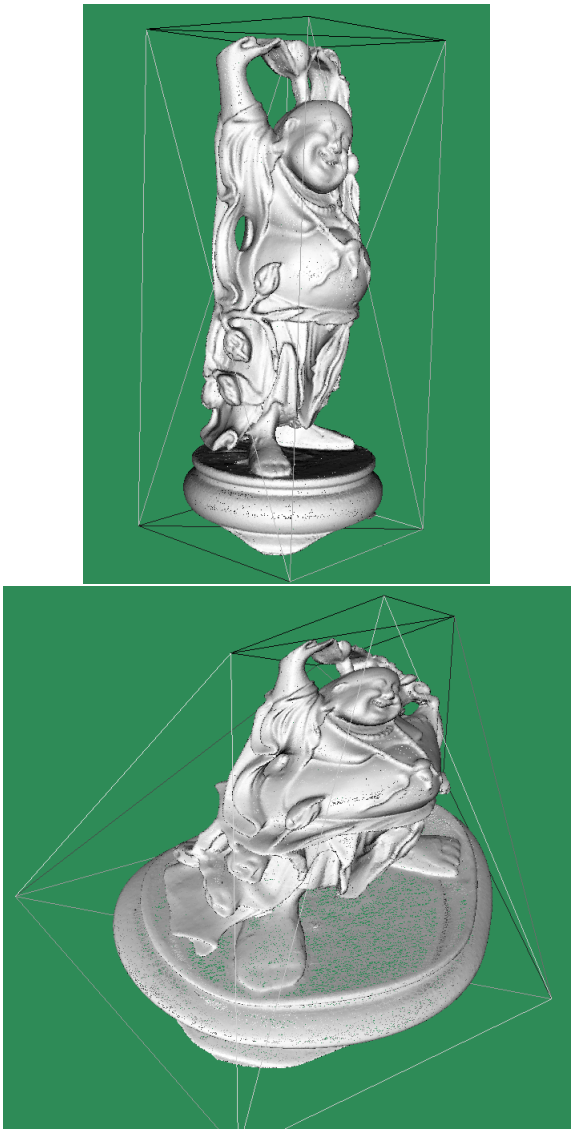


Figura 7.3: Happy Buddha deformat.

7.4 Temps a CPU i GPU

Les nostres proves consisteixen a mesurar el temps d'execució a CPU i GPU en el càlcul de les coordenades afins i el càlcul dels nous punts del model sobre els tres models. A la taula 7.1 s'hi troba el temps d'execució en ambdós casos.

Càlcul coordenades afins	CPU (msec)	GPU (msec)	Boost
Happy Buddha	2330	1344	73'59%
Stanford Bunny	156	18	766'67%
Camell	213	68	213'23%
Càlcul dels nous punts	CPU (msec)	GPU (msec)	
Happy Buddha	739	8	9137'5%
Stanford Bunny	48	2	2300%
Camell	53	2	2550%

Taula 7.1: Taula comparativa dels temps d'execució entre CPU i GPU.

7.5 Anàlisis dels resultats

El resultat en les deformacions són força bons en tots els models. En concret, el cas del Camell ha sigut el millor. El fet de disposar d'una caixa que s'adapti bé al model fa que la deformació sigui molt més detallada i acurada que no pas si el model s'envolta amb una caixa ortoedrica. Si mirem la figura 7.1, observem com hem pogut deformar només una part del model com ara cap, potes i gepa. En canvi, en els models restants es veu una deformació que engloba una gran part del model i no es focalitza en una part en concret.

Per altra banda, la fluïdesa amb que s'interactua i es deforma és molt bona en el Standford Bunny i el Camell. En canvi, en el Happy Buddha hem experimentat retards i lentitud. La nostra hipòtesis és que hi ha un coll d'ampolla en el el moment en que s'actualitzen els nous punts del model 3D. L'explicació és la següent: els nous punts calculats sobreescriuen als "antics" a memòria principal. En el moment en que s'han de visualitzar, es

transfereixen tots de memòria principal a la GPU a cada moviment del ratolí amb la qual cosa es transfereix molta informació en molt poc temps. Això fa que es produeixi un "embús" entre CPU i GPU que es fa més notori en models relativament grans com ara el Happy Buddha.

Pel que fa a la comparativa entre CPU i GPU, els resultats són positius. Veiem com en tots els casos, la GPU sempre és millor que la CPU. A l'apartat de "Càlcul coordenades afins" hi trobem el pitjor guany amb només un 73'59% per part del Happy Buddha. Cal destacar la regressió del guany que es dona entre Happy Buddha i Standford Bunny on l'única diferència entre ambdós és el nombre de punts ja que la caixa és la mateixa. A l'apartat "Càlcul dels nous punts", hi trobem els millors guanys. En particular, el millor és el Happy Buddha amb un 9137'5% d'increment. Destacar com, a diferència de l'apartat "Càlcul coordenades afins", s'incrementa el guany entre Happy Buddha i Stanford Bunny (recordem que tenen la mateixa caixa).

Capítol 8

Conclusions i treball futur

Gràcies a aquesta aplicació, hem comprovat que les mean value coordinates són una bona tècnica per deformar models 3D. Hem vist que és molt més interessant tenir una caixa específica d'un model per tal de fer les deformacions més acurades. Per desgracia, aquestes caixes no solen estar disponibles, fet que ha motivat utilitzar caixes més senzilles per tal d'eliminar aquesta restricció. També hem vist com, amb models grans, apareix el llast del coll d'ampolla en la transferència de dades entre CPU-GPU que fa alentir la visualització.

Quant a la implementació OpenCL a GPU, s'ha comprovat els extraordinaris beneficis que suposa. En tots els nostres resultats, hem obtingut millores importants en temps d'execució respecte una implementació tradicional a CPU. Això ens permet dir que la programació paral·lela és un avanç molt important cap a una millora en el rendiment del programari i serà un element comú i imprescindible en un futur no gaire llunyà en el desenvolupament d'aplicacions.

Quant al treball futur, una possible millora seria interessant estudiar la manera d'optimitzar i millorar la implementació GPU (kernels d'OpenCL). En especial, la part del càlcul coordenades afins.

Un altre possible treball, en aquest cas per millorar l'efecte de coll d'am-

polla comentat, és investigar la possibilitat d'incloure interoperabilitat entre OpenGL i OpenCL. Això implicaria que un búffer podria ésser compartit i utilitzat en ambdues plataformes de tal manera que faria innecessari la transferència de dades entre CPU i GPU. Per tant, tan els càlculs com la visualització es farien exclusivament a la GPU.

Capítol 9

Manual d'ús

En aquest capítol es pretén donar informació sobre com compilar i executar l'aplicació. També es donarà informació sobre quins paquets s'han d'instal·lar al sistema. Cal destacar que aquestes instruccions són orientades cap al sistema operatiu Ubuntu; tammateix, és altament probable que en altres distribucions Linux i/o UNIX siguin molt similar.

9.1 Programari necessari

Per executar l'aplicació, és necessari tenir els següents paquets d'Ubuntu instal·lats:

- Quant a compiladors:
  - build-essential
  - cmake
- Quant a llibreria VTK:
  - libvtk5.x
  - libvtk5-dev

A més a més, en cas que es vulgui utilitzar paral·lelisme (tant CPU com GPU), és necessari instal·lar els controladors i llibreries OpenCL d'acord amb el fabricant de la targeta gràfica o microprocessador del sistema. Aquests controladors es poden trobar fàcilment al web del fabricant.

9.1.1 Quant a CMake

CMake és un programari lliure multiplataforma per a la gestió del procés de construcció de programari utilitzant un mètode compilador independent de plataforma (cross-compiling). Està dissenyat per suportar jerarquies de directoris i aplicacions que depenen de diverses biblioteques, i per al seu ús en conjunció amb nadius d'entorns de compilació, com fer, Xcode d'Apple i Microsoft Visual Studio. També compta amb dependències mínimes, que només requereix un compilador C + + en el seu propi sistema de generació.

El procés de construcció es controla creant un o més fitxers CMakeLists.txt a cada directori (incloent subdirectoris). Cada CMakeLists.txt consisteix en un o més ordres. Cada comanda té la forma COMMAND (arguments, ...) on COMMAND és el nom de la comanda, i arguments és una llista d'arguments separats per espais.

Per construir un programa, tradicionalment és genera un fitxer Makefile que depèn de la plataforma i s'executa la comanda "make" per generar el fitxer executable. Doncs bé, el CMake automatitza el procés i genera automàticament el fitxer Makefile adient a la plataforma executant l'ordre "cmake <root directory>". D'aquesta manera, aconseguim molta portabilitat a l'hora de compilar i construir un programa en diverses plataformes.

9.2 Compilació

En aquesta secció es donaran les instruccions per compilar el còdi font. Aquest es troba dins de la carpeta "src".Per compilar, fem el següent:

Obrim un terminal, ens situem al directori on hi hagi "src" i creem una nova carpeta (p.e "build").

```
$ mkdir build
```

Ens situem dins d'aquesta nova carpeta.

```
$ cd build/
```

Generem el Makefile amb el CMake

```
$ cmake ../src/
```

Compilem

```
$ make
```

Si tot ha anat correctament, a dins de la carpeta build hi haurà el fitxer executable anomenat "meanval".

Cal destacar que si es vol executar sense OpenCL, s'ha de comentar la línia "#define OPENCL" del fitxer parallel.h i recompilar el còdi.

9.3 Execució

En aquesta secció s'explicarà el procediment per executar l'aplicació.

L'aplicació espera dos paràmetres d'entrada:

- El fitxer del model amb l'opció -m
- El fitxer de la caixa amb l'opció -c (Opcional)

Per executar l'aplicació, ens situem dins del directori que conté l'executable i fem:

```
$ ./meanval -m ''fitxer-del-model'' [-c ''fitxer-de-la-caixa'']
```

Un cop carregada l'aplicació, és recomanable pitjar la tecla 'w' per a activar el mode "wired" de la visualització per tal de veure la caixa i el model transparents. Per tornar al mode inicial, anomenat "solid", cal pitjar la tecla 's'.

Per deformar, ens situem sobre un vèrtex de la caixa, el pitgem amb el botó del mig del ratolí i el desplacem (mantenint el botó pitjat) on desitgem.

Per sortir de l'aplicació, podem tancar la finestra o pitjar la tecla 'q'.

Bibliografia

[1] JU T., SCHAEFER S., WARREN J.: Mean value coordinates for closed triangular meshes. Proc. SIGGRAPH 2005, vol. 24(3), pages. 561–566.

[2] LIPMAN Y., KOPF J., COHEN-OR D., LEVIN D.: Positive mean value coordinates. Eurographics Symposium on Geometry Processing 2007.

[3] JOSHI P., MEYER M., DeROSE T.: Harmonic coordinates for Character Articulation. Pixar Animation Studios.

[4] H ORMANN , K. 2004. Barycentric coordinates for arbitrary polygons in the plane. Tech. rep., Clausthal University of Technology, September. http://www.in.tu- clausthal.de/ hormann/papers/barycentric.pdf.

[5] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry y Dana Schaa (2011). Heterogeneous Computing with OpenCL. Morgan Kaufmann. ISBN: 9780124058941.

[6] OpenCL - The open standard for parallel programming of heterogeneous systems. Khronos Group. http://www.khronos.org/opencl/

[7] VTK - KitwarePublic. Kitware Inc. http://www.vtk.org/Wiki/VTK

[8] The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/