Facultat d'Informàtica de Barcelona

Sistema de Realitat Augmentada amb Kinect

Memòria del projecte

Emilio Alumne: Emilio Pastor Moreno
Director: Enric Xavier Martín Rull
Tardor 2011/2012

Contingut

1. Introducció 4
1.1. Què és la Realitat Augmentada? 4
1.2. Tecnologia 4
1.3. Aplicacions 5
2. Objectius del projecte 6
3. Esquema general del projecte 7
4. Controladors 8
4.1. SDK Microsoft 8
4.2. OpenNI 8
4.3. Elecció final 9
5. OpenNI 10
5.1. Què és OpenNI? 10
5.1.1. Vista abstracta de capes 10
5.2. Nodes de Producció 11
5.2.1. Tipus 12
5.2.2. Capacitats 12
5.3. Generació de dades 13
5.4. Lectura de dades 13
6. Processat de dades 15
6.1. Nodes de producció 15
6.1.1. Configuració de nodes 15
6.1.2. Preparació de l'aplicació 17
6.2. Extracció de dades 18
7. Model i Estructura de dades 20
7.1. Estructura de dades 20
7.1.1. Format del model 20
7.1.2. Diagrama de classes 21
7.1.3. Implementació de l'estructura de dades 24
7.2. Model 3D 26
7.2.1. Sketch Up 27
7.2.2. Model utilitzat 27
7.2.3. Lectura Model 28
7.3. Partició del model 29

7.3.1. Parts resultants 29
8. Visualització i resultats 33
8.1. Resultats parcials 33
8.2. OpenGL 35
8.3. Transformacions del model 38
9. Conclusions 39
10. Pressupost i planificació 40
11. Glossari de termes 42
12. Bibliografia 43

1. Introducció

1.1. Què és la Realitat Augmentada?

La Realitat Augmentada és el terme utilitzat per definir la combinació d'elements virtuals amb elements físics del món real de manera que es pugui crear una realitat mixta en temps real.

L'any 1994, es va definir la Realitat Augmentada com una zona intermèdia d'un espai fictici que uneix el món real i el món virtual.

Fig 1. Representació de Realitat augmentada

1.2. Tecnologia

En un sistema de Realitat Augmentada hi ha 4 blocs prou diferenciats, aquests blocs són la captura, la identificació, el processat i la visualització.

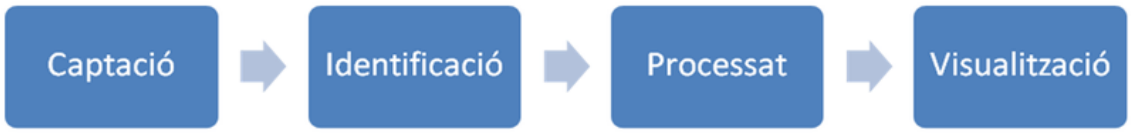


Fig 2. Blocs necessaris per construir un sistema de Realitat Augmentada

La **Captació** és la tasca encarregada de recollir l'escena del món real que es pretén augmentar. El dispositiu més utilitzat per aquesta tasca són les càmeres.

La **Identificació** consisteix en esbrinar quines parts del món físic que s'ha capturat es vol augmentar. Per tal de poder portar a terme el procés d'identificació de l'escena, hi ha dues tècniques: reconeixement utilitzant marcadors o sense marcadors.

El reconeixement utilitzant marcadors (*Marker Tracking*) està basat en la utilització de marcadors mitjançant els quals poder identificar la situació de l'escenari del món físic respecte al dispositiu de captura. A continuació es poden veure alguns exemples dels marcadors més estesos en el desenvolupament de sistemes de Realitat Augmentada.

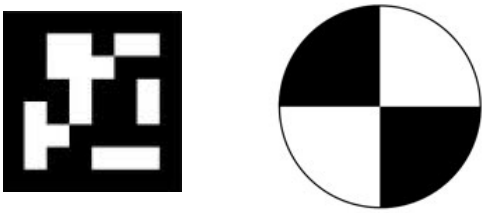


Fig 3. Exemples de marcadors utilitzats en Marker Tracking

Normalment, la localització dels marcadors dins de l'escena que ha s'ha capturat al bloc anterior té un cost computacional bastant elevat.

El reconeixement sense marcadors (*Markerless Tracking*) es basa en el reconeixement de l'escena sense marcadors. El seu sistema és més complex que el Marker Tracking degut a que no hi ha cap element auxiliar que permeti extreure'n dades de l'escena.

El **Processat** consisteix en combinar l'escena del món real tal com s'ha rebut a través del dispositiu de captura amb els elements virtuals que es volen afegir.

Finalment, la **Visualització** és la reproducció de l'escena real amb la informació virtual que se li afegeix. La representació dels objectes virtuals integrats en l'espai real no és un procés que resulti trivial, ja que s'ha d'haver realitzat un correcte reconeixement de l'entorn al analitzar la imatge captada pel dispositiu de captura.

1.3. Aplicacions

La Realitat Augmentada ofereix un ampli ventall de possibilitats d'interacció que fa que sigui possible trobar-ne sistemes en camps tant diversos com l'educació, la cirurgia, l'entreteniment, la publicitat, l'arquitectura o les retransmissions esportives.

2. Objectius del projecte

L'objectiu principal del projecte és aconseguir un sistema de Realitat Augmentada utilitzant la càmera Microsoft Kinect, basada un procés de llum estructurada.

Aquest tipus de sistema es basa en projectar sobre l'escena un patró conegut de punts i observar amb un dispositiu receptor la posició relativa a la que es veuen i la deformació dels patrons. En funció d'aquestes variacions de posició i forma dels patrons es pot calcular la profunditat de cada píxel de l'escena i l'orientació de cada punt.

Per aconseguir el sistema de realitat augmentada cal diferenciar diverses parts. Primer cal aconseguir la imatge real que rep la càmera, també cal aconseguir una imatge de profunditat i poder extreure'n d'aquesta última els usuaris que hi ha davant la càmera y la posició de les seves articulacions. Una vegada es té tota la informació sobre la posició dels usuaris, tenim moltes possibilitats, com per exemple, dibuixar un model 3D en la mateixa posició en la que es troba l'usuari o comprovar si amb l'últim moviment efectuat s'ha produït alguna interacció amb algun dels elements addicionals que proporciona el sistema de realitat augmentada.

Si mirem detingudament els blocs que formen part d'un sistema de Realitat Augmentada dels que es parlava a l'apartat anterior:

- La Captació la porta a terme la càmera utilitzada (Microsoft Kinect).
- La Identificació es porta a terme mitjançant el controlador utilitzat per aconseguir les dades de la càmera, com es veurà més endavant.
- Els dos últims blocs (Processat i Visualització) es porten a terme des de una aplicació creada ad hoc per al projecte, com es veurà al llarg d'aquesta memòria.

3. Esquema general del projecte

Com s'ha explicat a l'apartat anterior, el projecte consisteix en dos elements hardware: la càmera Microsoft Kinect i un ordinador portàtil des d'on es fa córrer el software necessari per aconseguir el sistema de Realitat Augmentada.

Pel que fa als elements software, es pot trobar el controlador capaç de rebre les dades procedents de la càmera i una aplicació desenvolupada per al projecte consistent en la utilització de les classes que proporciona el controlador per poder accedir a les dades rebudes de la càmera, el processat d'aquestes dades (detecció d'usuaris) i la visualització del món real amb els elements virtuals afegits pel sistema de Realitat Augmentada.

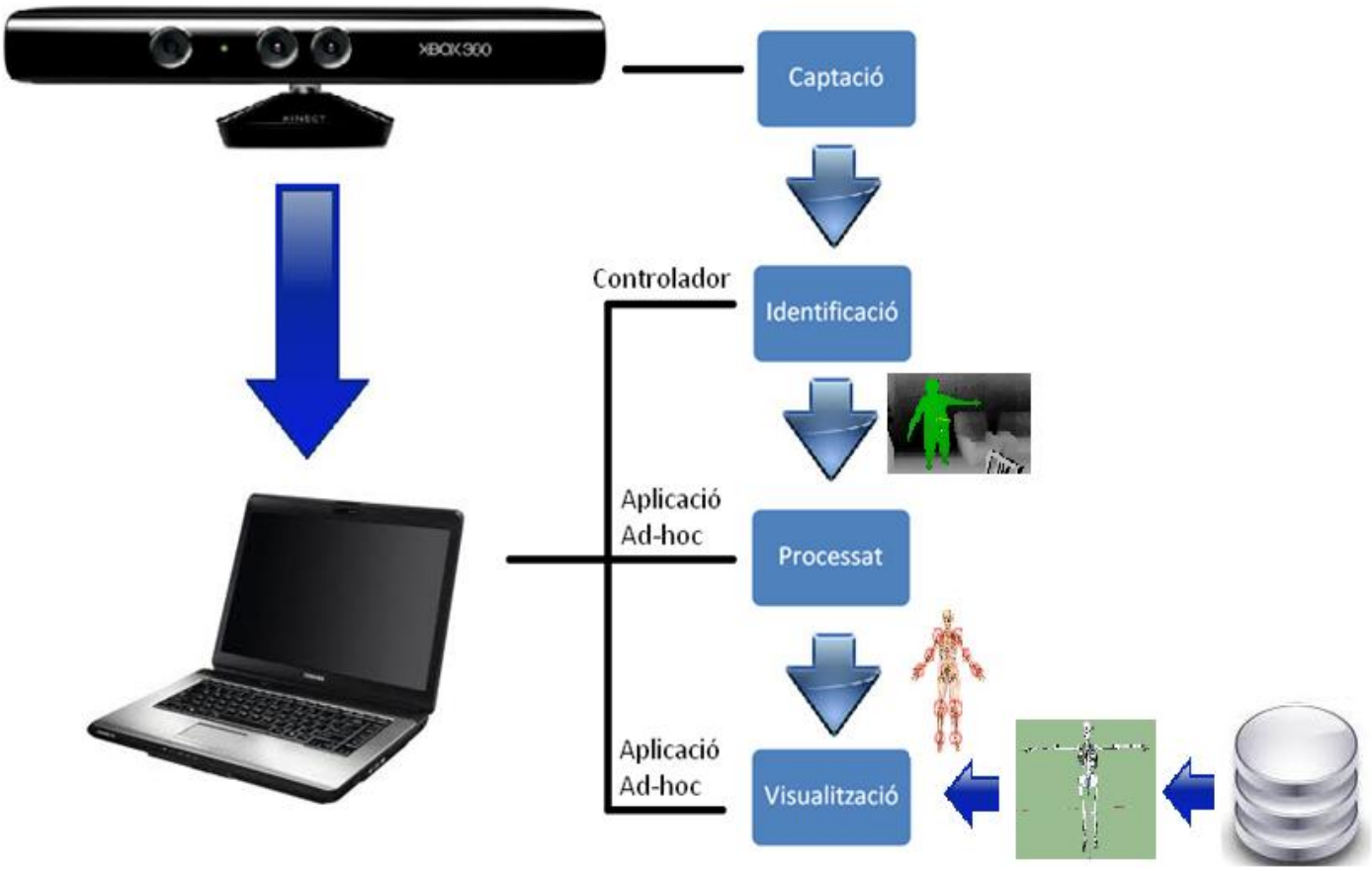


Fig 4. Esquema general del projecte

4. Controladors

Per aconseguir les dades que ens proporciona el Microsoft Kinect existeixen diferents controladors. En aquest apartat es pretén mostrar els controladors que s'han tingut en compte i justificar l'elecció final.

4.1. SDK Microsoft

El primer controlador que s'ha tingut en compte és l'SDK (Software Development Kit) oficial de Microsoft per al dispositiu Microsoft Kinect. Dels controladors analitzats aquest és sense cap dubte, el més conegut ja que és l'oficial de Microsoft i el que és més nou pel que fa a la data de sortida.

El controlador es pot descarregar sense cap problema des de la pàgina web oficial del controlador de Microsoft [1]. Abans d'instal·lar l'arxiu descarregat és necessari que l'equip on es vulgui executar l'aplicació que rebi les dades del dispositiu ha de tenir instal·lat també el Microsoft Visual Editor 2010 Express i el Microsoft .NET Framework 4.0.

Aquests dos paquets de software són molt extensos perquè no només serveixen per poder provar y desenvolupar aplicacions que funcionin amb el sensor Microsoft Kinect. Això fa que el temps d'instal·lació d'aquests dos paquets de software sigui excessiu, tot i que això no és rellevant una vegada s'ha instal·lat.

Per poder executar les proves amb tot el software que porta darrere són necessaris uns recursos hardware dels que no es disposàvem i excessius per la majoria dels cassos. És per això que a les proves que es van fer, els resultats de la visualització no van ser tan bons com s'esperaven. Tot i així es va poder comprovar que l'SDK funciona prou bé i va ser una opció clara com a controlador utilitzat per desenvolupar el sistema de Realitat Augmentada.

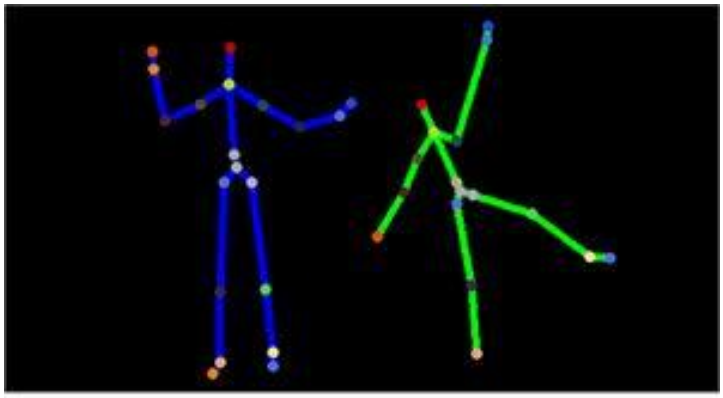


Fig 5. Exemple prova amb l'SDK oficial de Microsoft

4.2. OpenNI

Aquest controlador és un controlador de codi lliure desenvolupat per PrimeSense. Va sortir abans que el SDK oficial de Microsoft i, per tant, es poden trobar més exemples d'aplicacions que utilitzen aquest controlador.

S'han de descarregar els tres mòduls necessaris de la pàgina oficial de OpenNI [2]. Aquests tres mòduls son:

- **Sensor**, encarregat d'implementar la interacció entre el dispositiu hardware i l'aplicació.
- **OpenNI**, que és pròpiament el controlador amb tot el necessari per poder executar algunes proves que ja venen al mateix paquet.
- Finalment, **NITE** és un middleware de PrimeSense focalitzat en habilitar la interacció natural. Aquest últim mòdul conté la infraestructura algorísmica per la identificació d'usuaris, detecció de gestos, etc, així com l'etiquetat d'usuaris a l'escena.

La instal·lació és molt senzilla, és molt fàcil trobar a Internet una guia que expliqui pas a pas com instal·lar OpenNI a un sistema operatiu Linux, així com la resolució de problemes durant la instal·lació [3].

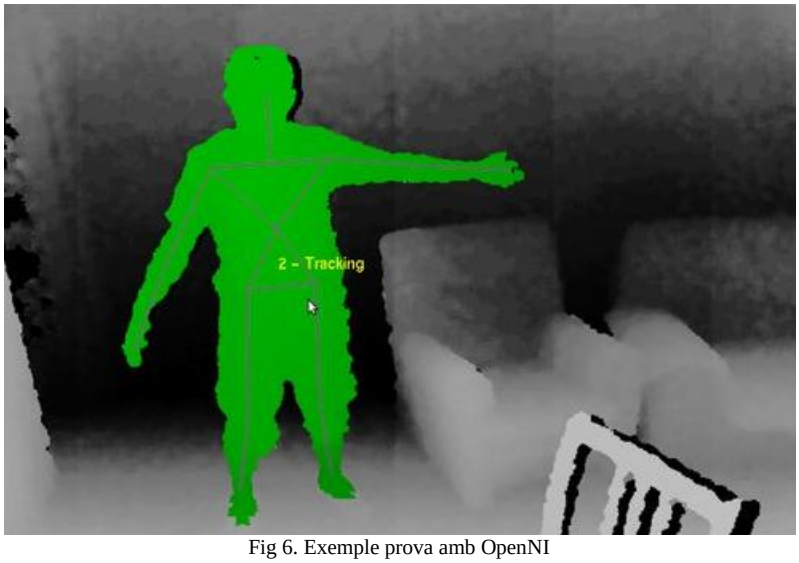


Fig 6. Exemple prova amb OpenNI

4.3. Elecció final

Tot i que tots els controladors que s'han mirat han donat resultats prou satisfactoris, s'ha escollit el controlador lliure OpenNI, per la facilitat que ens ofereix a l'hora de modificar i reutilitzar el codi dels samples que incorpora. D'aquesta manera és més fàcil apropar-nos a les classes y funcions que ens proporciona el controlador.

Com es descriu al següent apartat, el controlador escollit és fàcil d'entendre i dóna un conjunt molt interessant d'eines.

5. OpenNI

Com s'ha explicat en l'apartat anterior, s'ha utilitzat el controlador lliure OpenNI per la part d'identificació del sistema de Realitat Augmentada. En aquest apartat es pretén fer un estudi detallat del que ens ofereix OpenNI a l'hora de desenvolupar el nostre sistema.

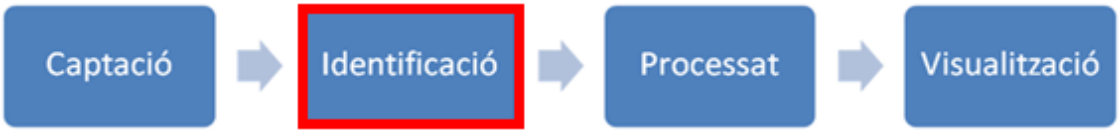


Fig 7. Blocs necessaris per construir d'un sistema de Realitat Augmentada

Ens trobem en el bloc que s'encarrega de la identificació dels diferents elements de l'escena i els prepara per que siguin processats posteriorment. Per aconseguir això, s'alimenta de les dades que s'han capturat anteriorment, provinents dels sensors que utilitzem.

5.1. Què és OpenNI?

OpenNI és un framework multi plataforma que defineix API's en diferents llenguatges de programació per poder desenvolupar aplicacions que utilitzin interacció natural (NI). El terme NI(*Natural Interaction*) prové de la interacció usuari-dispositiu basada en els sentits humans, com poden ser l'oïda o la visió. Alguns exemples d'interacció natural utilitzats el dia a dia són:

- Reconeixement de veu, el dispositiu rep instruccions mitjançant comandes de veu.
- Gestos de les mans, gestos predefinits de les mans són reconeguts i interpretats.
- Reconeixement del moviment corporal, tots els moviments del cos són analitzats i interpretats.

Les API's de OpenNI estan composades d'un conjunt d'interfícies per poder desenvolupar aplicacions utilitzant interacció natural. Des de OpenNI és possible comunicar-se tant amb sensors de àudio o vídeo (capten informació del món que els envolta) com amb components software que s'encarreguen de la percepció d'àudio o vídeo.

També permet als desenvolupadors de components software poder escriure algorismes independentment del sensor que ha produït les dades i permet als fabricants construir sensors que funcionin amb qualsevol aplicació compatible amb OpenNI.

5.1.1. Vista abstracta de capes

A la imatge es poden veure clarament diferenciades les tres capes del concepte que utilitza OpenNI per desenvolupar aplicacions amb interacció natural.

A la part més alta trobem les aplicacions que utilitzen l'OpenNI per tal d'implementar sistemes d'interacció natural. A la part més baixa es poden observar els dispositius hardware encarregats de captar les dades del món físic. Al mig de la imatge es troba l'OpenNI, les seves

interfícies per comunicar-se amb els components hardware i software i també amb els Middleware's que encapsulen funcions específiques de la interacció natural, com ara el reconeixement d'usuaris (com veurem més endavant) o el reconeixement de les mans.

Podem veure que aquest esquema encaixa perfectament amb el sistema de Realitat Augmentada que s'ha plantejat als apartats 2 i 3 d'aquest document.

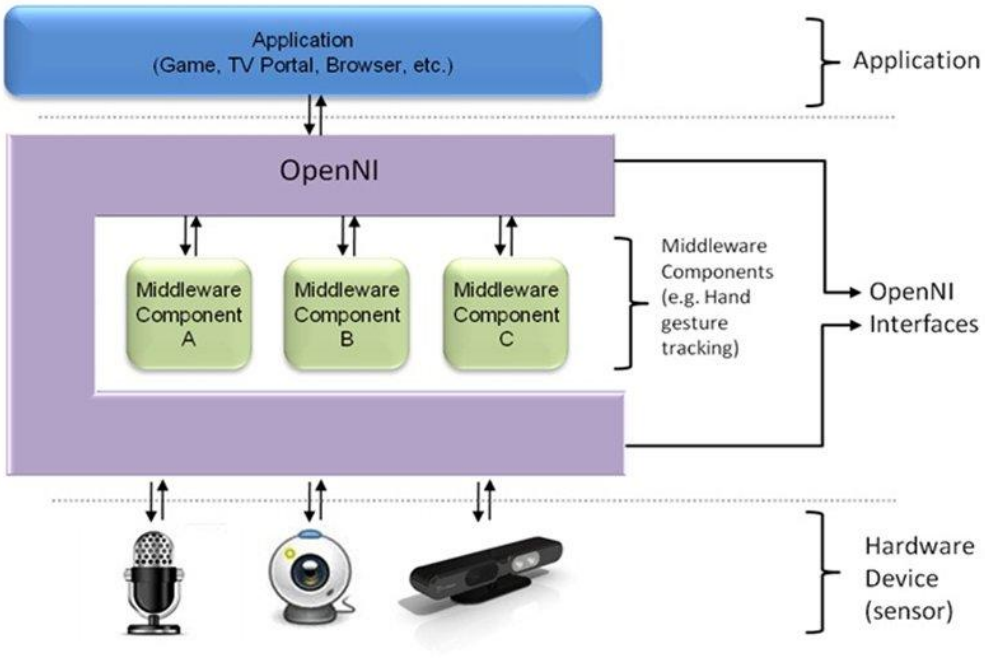


Fig 8. Esquema general de l'arquitectura de OpenNI

5.2. Nodes de Producció

OpenNI defineix nodes de producció, són un conjunt d'unitats que tenen un rol productiu en la creació de dades per una aplicació basada en interacció natural. Un node de producció qualsevol pot utilitzar altres nodes de producció més bàsics per llegir les seves dades o controlar la seva configuració, per exemple. De la mateixa manera, poden ser utilitzats per altres nodes de producció més complexos o directament per l'aplicació basada en interacció natural.

Entenem per dades 3D significatives les dades que som capaços de comprendre, entendre i traslladar a l'escena. Aquestes dades són difícils d'extreure directament de la informació que capten els sensors i normalment es necessària la utilització d'altre components o nodes més complexos per tal d'entendre el significat de les dades capturades.

Per exemple, es pot utilitzar un node de producció que generi un mapa de profunditats on, per cada píxel de la matriu resultant, el que tenim és la distància a la que es troba l'objecte que s'hi veu. Però si el que volem és saber la posició de les mans dels usuaris dins de l'escena, caldrà un altre node de producció més complex que agafarà el mapa de profunditats generat pel primer node més simple.

5.2.1. Tipus

Els nodes de producció que utilitza OpenNI tenen un tipus i pertanyen a una determinada categoria depenent de la seva complexitat. Les dues categories són nodes de producció relatius a sensors o nodes de producció relatius a un Middleware.

Els diferents tipus de nodes de producció relatius a sensors als que OpenNI dóna suport són:

- Device: és un node que representa un dispositiu físic. La seva principal funció és habilitar o no la configuració d'un dispositiu determinat.
- Depth Generator: és un node que genera mapes de profunditat.
- Image Generator: és un node que genera mapes d'imatges a color, és a dir, la imatge tal qual la veuria una càmera convencional.
- IR Generator: és un node que genera mapes d'imatges d'infrarojos.
- Àudio Generator: és un node que genera un flux d'àudio.

Qualsevol sensor 3D que es consideri compatible amb OpenNI suporta tots els tipus de nodes de producció que acabem de descriure.

Els diferents tipus de nodes de producció relatius a un Middleware que suporta OpenNI són:

- Gestures Alert Generator: genera callbacks a l'aplicació quan identifica gestos específics.
- Scene Analyzer: analitza l'escena, identifica les figures que n'hi ha i detecta el pla del terra. La principal sortida d'aquest tipus de node de producció és un mapa de profunditats etiquetat, on cada píxel té una etiqueta que indica si representa a una figura o forma part del fons de l'escena.
- Hand Point Generator: suporta la detecció i seguiment de les mans que es troben a l'escena. Aquest tipus de node de producció, genera callbacks a l'aplicació quan es detecta un punt d'una nova mà o quan una mà de la que s'està fent el seguiment canvia de posició.
- User Generator: genera una representació d'un cos en l'escena 3D.

5.2.2. Capacitats

Les mecanisme de capacitats d'OpenNI permet flexibilitat en el registre de diversos components software (Middleware) i dispositius.

Es pot donar el cas en que diferents elements necessitin dades del mateix node de producció però amb una configuració diferent. Per tant, l'API de OpenNI defineix les extensions no obligatòries que té un node de producció. Aquestes extensions no obligatòries s'anomenen **capacitats** i afegeixen funcionalitats addicionals, ja que es permet que qui utilitza el node de producció sigui capaç de decidir si vol que s'utilitzi una capacitat concreta o no.

Les principals capacitats suportades actualment per OpenNI són:

- Alternative View: habilita qualsevol tipus de generador de mapes per transformar les seves dades per que sembli que el sensor està situat a una altra posició, ja siguin de profunditats, imatges a color, imatges d'infrarojos, etc. Normalment, la posició alternativa està representada per un altre sensor.
- Cropping: permet que un generador de mapes tingui com a sortida un area determinada de l'escena que no sigui l'escena sencera. Es molt útil quan es busca una millora en l'eficiència, ja que permet explorar únicament la part de l'escena que ens interessa.
- Frame Sync: permet que dos sensors que produeixen informació visual de l'escena es sincronitzin de manera que les dades que capten estiguin disponibles a la vegada.
- Mirror: permet aconseguir l'efecte mirall a les dades produïdes pel generador. Es una capacitat molt útil pels casos en que el sensor es troba al davant de l'usuari.
- Pose Detection: permet que el generador d'usuaris reconegui quan l'usuari està en una posició determinada.
- Skeleton: habilita al generador d'usuaris per tenir com a sortida dades sobre l'esquelet de l'usuari. Per exemple, es podria fer que generés dades sobre la posició de les articulacions dels usuaris.

5.3. Generació de dades

Els nodes de producció que produeixen dades s'anomenen també generadors. Una vegada s'han creat no comencen a generar dades de forma immediata, per permetre especificar la configuració desitjada. Això assegura que, una vegada s'ha començat a generar dades, s'estan generant de la manera desitjada, donada la configuració. Per començar la generació de dades s'utilitza la funció *xn::Generator::StartGenerating()*.

Podria ser que l'aplicació volgués parar la generació de dades sense destruir el node que les està generant. Per exemple, per guardar la configuració que s'està utilitzant actualment. Per parar la generació de dades s'utilitza la funció *xn::Generator::StopGenerating()*.

5.4. Lectura de dades

Els generadors de dades constantment estan rebent noves dades, però podria ser que l'aplicació volgués utilitzar les dades del pas anterior. Per permetre això, els generadors de dades es guarden internament les noves dades fins que se li demana que actualitzi les dades amb les més noves de les que disposi.

Amb això volem dir que els generadors amaguen les noves dades que obtenen fins que algú se les demana utilitzant una funció d'actualització. OpenNI permet a l'aplicació esperar a que hi hagi noves dades i actualitzar-les tan bon punt n'hi hagi mitjançant la funció *xn::Generator::WaitAndUpdateData()*.

En determinats casos, l'aplicació manté diversos nodes de producció i es vol que s'actualitzin tots alhora. OpenNI proporciona diverses funcions que s'encarreguen d'esperar i actualitzar els nodes de producció de l'aplicació. Depenent del que es vulgui aconseguir, hi ha diferents funcions per aconseguir això. Totes aquestes funcions tenen un timeout als 2 segons:

- xn::Context::WaitAnyUpdateAll(): espera a que qualsevol node tingui dades noves. Una vegada hi ha dades disponibles a qualsevol dels nodes, actualitza les dades de tots.
- xn::Context::WaitOneUpdateAll(): espera a que un node determinat tingui noves dades disponibles. Una vegada hi ha noves dades, els actualitza tots. Aquesta opció és molt útil quan un únic node és l'encarregat de marcar el progrés de l'aplicació.
- xn::Context::WaitNoneUpdateAll(): no espera que hi hagi dades disponibles, en el mateix moment en que es crida aquesta funció, s'actualitzen totes les dades de tots els nodes de producció de l'aplicació.
- xn::Context::WaitAndUpdateAll(): espera fins que tots els nodes de l'aplicació tinguin noves dades disponibles. Actualitza les dades de tots una vegada tots tenen dades disponibles.

6. Processat de dades

En aquest apartat es pretén explicar detalladament la part de l'aplicació que s'ha desenvolupat per tal de suportar el bloc de Processat del sistema de Realitat Augmentada.

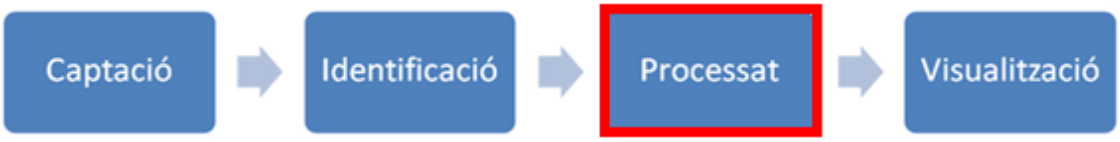


Fig 9. Blocs necessaris per construir un sistema de Realitat Augmentada

Ens trobem en el tercer bloc dels necessaris per la construcció d'un sistema de Realitat Augmentada. S'agafen les dades que s'han identificat anteriorment i es processen per tal de conèixer l'estat de l'escena.

6.1. Nodes de producció

Per poder desenvolupar el nostre sistema de Realitat Augmentada s'ha de definir a un fitxer XML de configuració quins nodes es volen utilitzar i les capacitats que volem que tinguin. Per poder utilitzar-los, els primers que fa l'aplicació és comprovar si aquests nodes estan declarats a l'arxiu XML. Registra un conjunt de callbacks que ens ajudaran l'estat en que es troba el seguiment dels usuaris i també fa diverses comprovacions relatives a les mides i el format de les imatges que es reben. Totes les passes que fa l'aplicació s'explicaran detingudament en aquest apartat.

6.1.1. Configuració de nodes

Tal com s'ha explicat, per configurar els nodes de producció necessaris s'ha de declarar quins nodes s'utilitzaran i amb quines capacitats.

En el nostre cas, l'arxiu de configuració XML no és gaire extens. La part més rellevant és la declaració dels nodes de producció:

```
<ProductionNodes>
        <GlobalMirror on="true"/>
        <Node type="Depth" name="Depth1">
        </Node>
        <Node type="Image" name="Image1" stopOnError="false">
        </Node>
        <Node type="User" name="User1" stopOnError="false">
        </Node>
</ProductionNodes>
```

Es pot veure com s'han declarat tres nodes de producció, tots tres de tipus diferents: *Depth*, *Image* i *User*.

Pel que fa a les capacitats definides pels nodes, només s'ha hagut de declarar una: *Mirror*. A més, aquesta capacitat s'ha declarat de forma global perquè volem que tots els

nodes de producció produeixin dades tals que, a l'hora de mostrar-li a l'usuari, provoqui l'efecte mirall. L'efecte final és com si l'usuari es trobés davant d'un mirall que no mostra la realitat sinó una virtual que li és semblant. D'aquesta manera s'aconsegueix una adaptació més ràpida de l'usuari a la visualització que se li està mostrant.

El node de producció de tipus User és un node especial, això ve donat perquè no és un node que rebi dades de cap sensor, sinó que utilitza dades d'altres nodes de producció per tal d'aconseguir les dades que vol oferir. Per aquest mateix motiu, les capacitats del node User no s'han hagut de definir al fitxer de configuració, són capacitats que ja venen definides pel propòsit que té el node de producció i no tindria sentit que no estiguessin. Les capacitats ja definides al node de producció de tipus User són *Pose Detection* i *Skeleton*.

La capacitat Pose Detection serveix per fer que la detecció de l'usuari es faci quan l'usuari adopta una posició determinada. A següent imatge es pot veure la posició necessària per que el node de producció detecti la posició de l'usuari.

Fig 10. Posició amb la que es detecta
la posició de l'usuari

La capacitat Skeleton serveix per poder saber en quines posicions es troben les articulacions de l'usuari una vegada s'ha detectat la seva posició. Una vegada es té la posició de les articulacions es pot començar a especular sobre el seu moviment (tenint en compte la posició anterior) o visualitzar aquestes posicions, com es veu a la següent imatge.

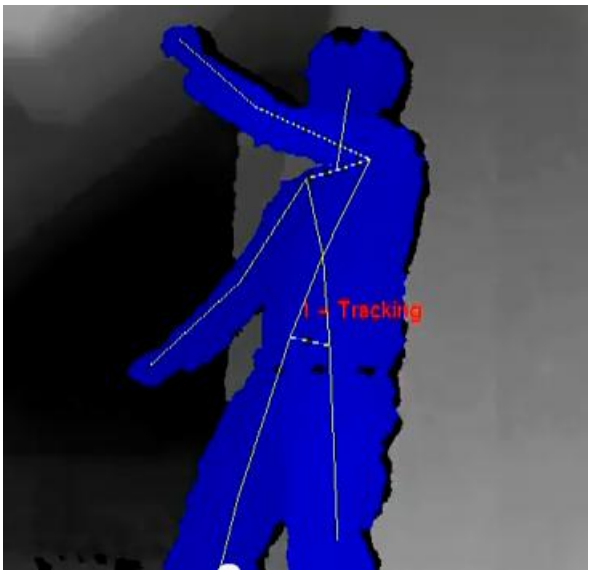


Fig 11. Visualització de l'esquelet dibuixat
amb línies una vegada detectat l'usuari

De la visualització que es fa una vegada s'aconsegueix es tenen les articulacions de l'usuari se'n parlarà detingudament més endavant.

6.1.2. Preparació de l'aplicació

Quan parlem de preparació de l'aplicació ens referim a tots els passos que fa l'aplicació abans d'entrar al bucle infinit que suposa el nucli de l'execució de l'aplicació.

El primer de tot que fa l'aplicació és inicialitzar tot el framework OpenNI des del fitxer XML de configuració mencionat prèviament. Utilitza la funció *xn::Context::InitFromXmlFile*(). Aquesta funció retorna un codi que indica si s'ha produït algun error a l'hora de carregar la configuració. Es comprova si l'error és XN_STATUS_NO_NODE_PRESENT, que ens indica que algun dels tipus de nodes declarats al fitxer de configuració no existeix. També es comprova si el codi és diferent de XN_STATUS_OK, ja que això significaria que la càrrega de la configuració ha fallat per alguna causa que desconeixem.

El següent pas és comprovar que els nodes de producció que volem utilitzar han sigut definits al fitxer de configuració. S'utilitza la funció *xn::Context::FindExistingNode*(), que rep com a paràmetre el tipus de node que es vol utilitzar i la variable que hem declarat per poder-nos referir a aquest node. La funció retorna un codi d'error igual que la funció anterior i es comprova que no sigui diferent de XN_STATUS_OK en cap dels tres casos.

Les variables amb les que ens referim als nodes de producció que es volen utilitzar són d'un tipus definit pel mateix OpenNI:

```
DepthGenerator g_depth;
ImageGenerator g_image;
UserGenerator g_user;
```

A continuació, es registren alguns callbacks utilitzats per escriure missatges per consola que ens ajudin a saber l'estat en que es troba en tot moment la detecció dels usuaris. D'aquesta manera obtenim missatges informatius com els de la següent imatge.

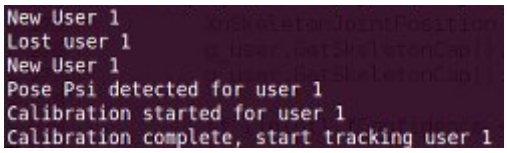


Fig 12. Exemple de missatges escrits per consola

Una vegada s'han registrat tots els callbacks, tots els nodes estan preparats i es comença a generar dades.

Queda comprovar que la mida i el format dels diferents mapes que s'estan generant són els esperats. Per això s'accedeix a les meta dades dels nodes generadors de profunditats i imatges a color.

```
DepthMetaData g_depthMD;
ImageMetaData g_imageMD;
g_depth.GetMetaData(g_depthMD);
g_image.GetMetaData(g_imageMD);
```

Des de les meta dades som capaços d'accedir al format de les imatges a color (volem que el format dels píxels sigui RGB24) i a la mida que tenen els mapes que s'estan generant (volem que els mapes de profunditats i d'imatges a color tinguin les mateixes mides).

6.2. Extracció de dades

Des de les meta dades de les que es parlava a l'apartat anterior som també capaços d'accedir als mapes que s'estan generant.

Una vegada al codi del bucle infinit que suposa el gruix de l'execució de l'aplicació. A cada volta del bucle infinit es crida la funció que actualitza tots els nodes de producció que estan generant dades, en el nostre cas són DepthGenerator i ImageGenerator.

Quan ja es tenen dades disponibles, s'actualitzen les meta dades dels dos generadors que reben dades de sensors.

```
g_depth.GetMetaData(g_depthMD);
g_image.GetMetaData(g_imageMD);
const XnDepthPixel* pDepth = g_depthMD.Data();
const XnUInt8* pImage = g_imageMD.Data();
```

A més, s'utilitza un altre tipus de meta dades (meta dades de l'escena) per extreure les dades del generador d'informació d'usuaris.

```
SceneMetaData g_sceneMD;
g_user.GetUserPixels(0,g_sceneMD);
const XnLabel* pLabels = g_sceneMD.Data();
```

```
XnUserID aUsers[15];
XnUInt16 nUsers = 15;
g_user.GetUsers(aUsers, nUsers);
```

Una vegada executat aquest codi com capaços d'obtenir els píxels que pertanyen a un usuari determina, l'usuari concret al que pertany ho indica una etiqueta assignada a cada píxel. Per una altra banda, també som capaços d'aconseguir informació sobre els usuaris a un vector de XnUserID i el nombre d'usuaris que hi ha actualment davant del sensor. De les variables de tipus XnUserID s'extreu més tard la posició de cada articulació dels diferents usuaris mitjançant la funció *g_user.GetSkeletonCap().GetSkeletonJointPosition*().

7. Model i Estructura de dades

En aquest apartat y el següent es pretén explicar detalladament la part de l'aplicació que s'ha desenvolupat per tal de suportar el bloc de Visualització del sistema de Realitat Augmentada.

Fig 13. Blocs necessaris per construir un sistema de Realitat Augmentada

Ens trobem a l'últim bloc dels necessaris per construir el sistema de Realitat Augmentada, el que ens proporcionarà el resultat que finalment rebrà l'usuari.

Es vol pintar un model 3D a la posició e al que es troba l'usuari, d'aquesta manera no es veurà a sí mateix sinó a un model 3D que es mou exactament igual a com ho fa l'usuari. Per tal d'aconseguir això, s'ha de dissenyar i implementar una estructura de dades que suporti el format en que ens arriba el model 3D, escollir un model i programar la visualització.

7.1. Estructura de dades

Per tal de carregar un model 3D, hem de tenir preparada una estructura de dades que suporti el format en què ens arriba el model.

7.1.1. Format del model

La primera decisió que s'ha d'afrontar en el bloc de Visualització consisteix en escollir un format en què aconseguim el model 3D que volem mostrar. Busquem que sigui fàcil d'aconseguir i fàcil de parcejar. Per que el model es mogui de la mateixa manera que ho fa l'usuari, voldrem que el model estigui articulat, pel que necessitarem que tingui un format que ens permeti tractar-lo d'alguna manera.

Els formats que es van mirar detingudament van ser:

- .3ds, és el format utilitzat pel software Visual 3D Studio. Aquest software és molt conegut i el format està prou extens com per trobar fàcilment un model en aquest format. Va ser una opció a tenir en compte des de bon començament.
- .max: no és tan conegut com .3ds però es fàcil aconseguir models en aquest format. Tot i així, no era gaire clar com fer un carregador del model per quan l'aplicació necessités les dades del model.
- .mb: no està gaire estès i no és gens fàcil trobar models en aquest format. Té el mateix problema que .max a l'hora de carregar el model.
- .skb: és el format utilitzat pel software Google SketchUp. Aquest software és prou conegut i la seva versió Pro permet exportar a una gran quantitat de formats. També

és fàcil aconseguir models en aquest format donat que el mateix software ofereix una llibreria en què es poden trobar models 3D de qualsevol tipus.

- .obj: és un format molt estès, les dades es codifiquen en text pla al document, pel que és fàcil fer un parser que ho llegeixi de forma seqüencial per tal de carregar totes les dades.

La decisió final va ser dividir l'obtenció del model de la càrrega:

1. Utilitzem el software Google SketchUp 8 Pro per descarregar el model 3D que vulguem en format .skb.
2. A continuació dividim el model en diferents parts per tal de poder tenir-lo articulat posteriorment. Més tard s'especifica quines parts són les resultants.
3. Exportem cadascuna de les parts obtingudes a format .obj.
4. L'aplicació s'encarrega de llegir cadascun dels fitxers resultants d'exportar des de SketchUp i els carrega de manera que es tinguin totes les dades per poder pintar el model 3D.

7.1.2. Diagrama de classes

Per suportar les dades que arriben en format .obj, s'ha de dissenyar com es voldrà tenir en memòria les dades del model 3D. Es va fer un diagrama de classes en UML amb el software Dia per tal de tenir especificat com havia de ser l'estructura de dades. D'aquesta manera, es facilita la posterior implementació de les classes obtingudes, ja que la traducció d'un diagrama de classes UML a codi és prou mecànica com per que no s'hagin de prendre decisions importants.

Com es veurà a continuació, s'ha dissenyat de forma descendent, és a dir, es comença amb una classe que conté unes altres i així seqüencialment fins arribar a les més bàsiques. D'aquesta manera, la nostra aplicació només interactua amb una classe referent al model, tot el tractament intern que es fa al model és invisible per l'aplicació. Amb això s'aconsegueix que el model sigui força canviable, és a dir, que un canvi la forma de representar les dades no suposi canvis en l'aplicació que s'ha desenvolupat.

Gràcies al disseny canviable que s'ha fet per l'estructura de dades, l'aplicació només ha de conèixer dues funcions de la classe Model que ha de cridar quan les necessiti, a mode d'interfície:

- *Model::readModel()*: és la funció encarregada de carregar totes els parts del model i tots els elements addicionals que pugui necessitar.
- *Model::drawObject()*: és la funció encarregada de pintar un objecte determinat. Abans de crida aquesta funció, l'aplicació només ha de preocupar-se de situar la càmera al VRP (View Reference Point) i aconseguir les dades de les articulacions que delimiten la part del model que es vol pintar.

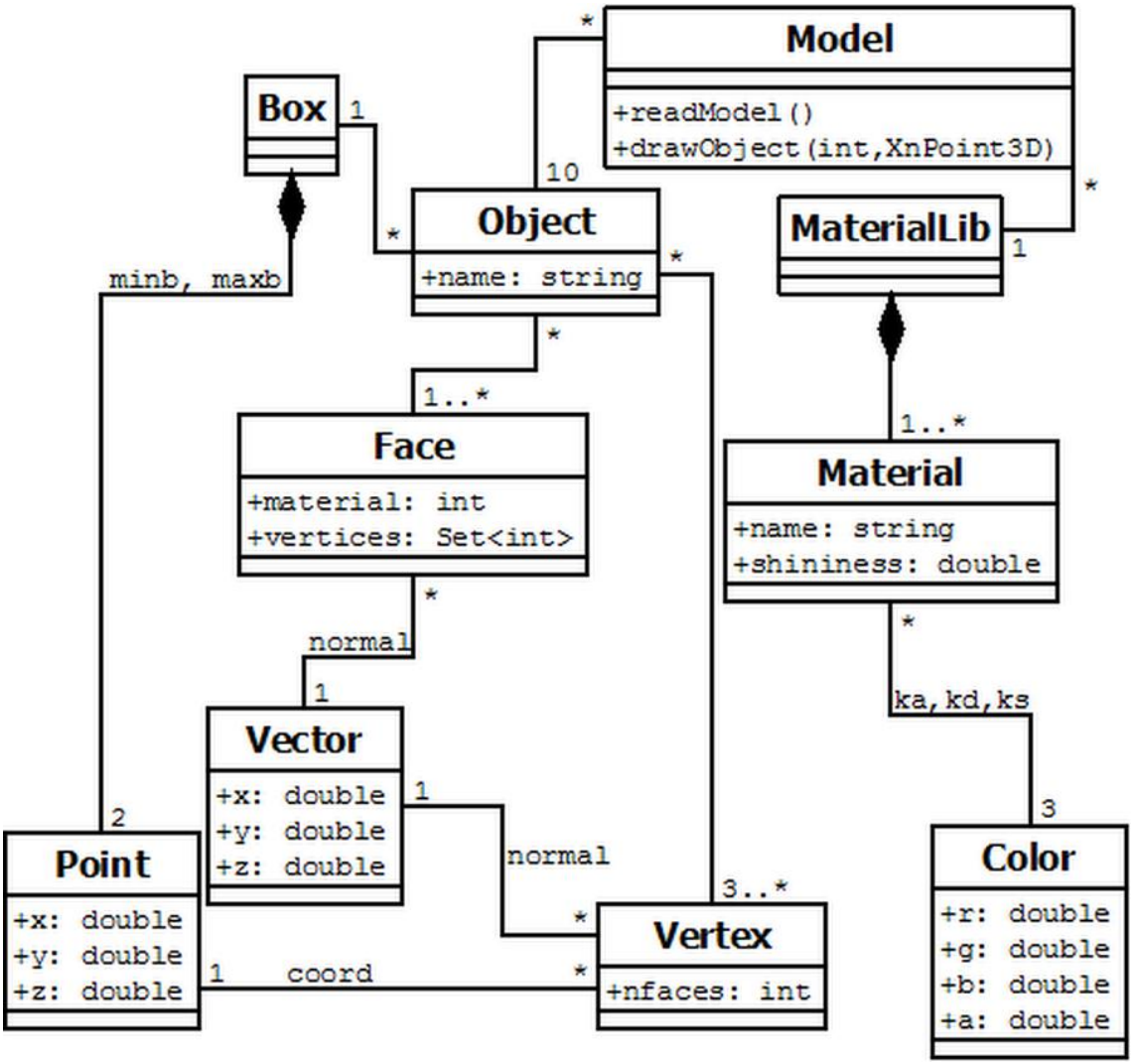


Fig 14. Diagrama de classes en UML de l'estructura de dades dissenyada

Ja s'ha començat a explicar la classe *Model*, no té cap atribut de tipus simple, però sí que conté una llibreria de materials, de la classe MaterialLib, i 10 objectes de la classe Object. Aquest 10 objectes de la classe Object contenen la informació de cadascuna de les parts en les que es parteix el model.

La classe *MaterialLib* és una composició de materials, per representar cadascun d'aquests materials s'ha creat la classe Material.

La classe *Material* conté el nom del material, provinent del fitxer .obj, un atribut que representa la brillantor de l'objecte, shininess. També conté tres objectes de la classe Color, com es pot veure etiquetat a l'associació:

- ka representa la llum ambient
- kd representa la llum difosa
- ks representa la llum especular

Més tard es parlarà detingudament de la funció de cadascuna de les llums utilitzades.

La classe *Color* conté 3 atributs de tipus double que representen la codificació RGB (r pel vermell,g pel verd i b pel blau), a més de l'atribut a, necessari per completar la codificació i fer la crida a glColor(), com s'explicarà més endavant.

La classe *Object* conté un atribut corresponent al nom de l'objecte, un objecte de la classe Box corresponent a la caixa contenidora de l'objecte, és a dir, una caixa que conté tots els punts de l'objecte. Aquest atribut ens servirà per poder dimensionar l'objecte posteriorment. També conté dues col·leccions d'objectes. La primera col·lecció és d'objectes de classe Vertex i representa tots els punts que són vèrtexs d'alguna cara de l'objecte. La segona col·lecció és d'objectes de la classe Face, que representa totes les cares que formen part de l'objecte.

La classe *Face* conté un enter com a identificador del material del que està fet la cara, s'utilitza per accedir a la llibreria de materials i aconseguir la informació del material. També conté un conjunto d'enters que són identificadors dels vèrtexs que formen part d'aquesta cara. També conté un objecte de la classe Vector, que representa la normal de la cara, en cas que es vulgui fer la visualització utilitzant normals per cares, com s'explicarà més endavant.

La classe *Vertex* conté un enter que indica el número de cares en què conflueix, aquest atribut serveix per quan es fa el càlcul de la normal del vèrtex. També conté un objecte de la classe Point per poder representar les coordenades del vèrtex i un objecte de la classe Vector per representar la normal del vèrtex, que s'utilitza en cas que es vulgui fer la visualització utilitzant normals per vèrtex.

La classe *Box* és una composició de dos objectes de la classe Point que indiquen les coordenades dels dos extrems oposats de la capsa contenidora. Un per les coordenades amb x,y i z mínimes i un altre per les coordenades amb les x,y i z màximes.

La classe *Point* conté tres atributs de tipus double que representen les coordenades del punt en l'espai (x,y,z).

Tal com passa a la classe Point, la classe *Vector* conté tres atributs de tipus double que representen les tres components x,y,z del vector.

Les classes Point i Vector contenen els mateixos atributs i es podrien haver fusionat en una sola, però d'aquesta manera queda més clar què és el que representa cada classe.

7.1.3. Implementació de l'estructura de dades

En aquest apartat es parlarà de les principals classes que conté l'estructura de dades i la declaració de les mateixes.

Com s'ha dit anteriorment, s'ha dissenyat de forma descendent, cosa que ens permet que per representar les associacions els elements de més amunt continguin com a atributs els elements de sota.

A la implementació de la classe Model queda prou clar quina és la idea:

```
class Model {

    private:
        Object cabeza;
        Object tronco;
        Object antebrazo_izqdo;
        Object antebrazo_drcho;
        Object brazo_izqdo;
        Object brazo_drcho;
        Object antepierna_izqda;
        Object antepierna_drcha;
        Object pierna_izqda;
        Object pierna_drcha;

    public:
        MaterialLib matlib;

        Model();
        void readModel();
        void drawObject(int num, XnPoint3D pt[]);

};
```

Es poden veure tots els elements de la classe Object que representen les diferents parts del model i la llibreria de materials. També podem observar la declaració de les funcions necessàries d'aquesta classe, són aquelles que s'utilitzen a mode d'interfície, com s'ha dit anteriorment. Trobem el constructor *Model()*, *readModel()* i *drawObject()*, a la que se li passa com a paràmetre un enter que codifica l'objecte que es vol pintar i una col·lecció d'objectes de la classe XnPoint3D, que és una classe de OpenNI que codifica la posició de les articulacions que li volem passar.

L'altra classe rellevant de la nostra estructura de dades és Object. Si mirem el seu codi podem veure les funcions constructores, la que s'utilitza per llegir l'objecte, la que s'utilitza per pintar l'objecte (Render()), la que calcula les normals de l'objecte i les que utilitzen actualitzen i retornen la capsa envoltant de l'objecte (Bounding Box).

Si mirem a les funcions privades que utilitza, són aquelles a les que és necessari crida quan s'està llegint l'objecte, aquestes funcions són *make_face()* i *netejaDades()*.

També es pot veure la declaració dels camps en els que emmagatzemem la informació de les cares i els vèrtexs que formen part de l'objecte.

```
class Object
{
 private:
  void make_face ( char **words, int nwords, int material );
  void netejaDades ();

  string name;
  Box _boundingBox;

 public:
  Object();
  Object(std::string);
  void readObj(string name, MaterialLib& matlib);
  void Render();
  void CalculaNormals();
  void updateBoundingBox();
  Box getBoundingBox();

  vector<Vertex> vertices;  // vector amb els vertexs de l'objecte
  vector<Face> faces;       // vector amb les cares de l'objecte
};
```

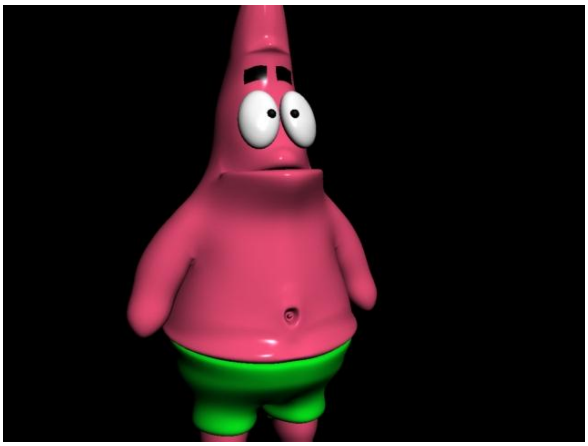
Cal fer menció també de la classe MaterialLib, que s'encarrega de llegir i emmagatzemar la llibreria de materials que s'utilitzarà. Al codi següent es pot veure la declaració de la funció *readMtl()*, que s'encarrega de llegir el fitxer que conté els materials que s'utilitzaran. La funció *material()* retorna un material donat un índex i la funció *index()* retorna l'índex del material donat el seu nom. Es pot observar també que es manté un diccionari per fer més fàcil l'accés als índexs donat el nom d'un material.

```
class MaterialLib
{
  public:
    MaterialLib();
    void readMtl(const char* filename);
    const Material& material(int index) const;
    int index(const string& name) const;

  private:
    vector<Material> materials;
    mutable map<string, int> dict;

    friend ostream& operator<< (ostream &os, const MaterialLib &m);
};
```

7.2. Model 3D

Com s'ha explicat anteriorment, s'ha instal·lat el software Google SketchUp 8 Pro i s'ha buscat un model a la llibreria de models que ofereix aquest software.

Abans de decidir utilitzar aquest software per buscar i tractar el model 3D, s'havien mirat altres models i formats. A continuació es mostren alguns dels models que al final es van descartar per tenir un altre format, necessitar un altre software que no teníem disponible o problemes a l'hora de partir el model per tal de poder articular-lo.



FIgs 15,16 i 17. Exemples de models 3D tinguts en compte abans de seleccionar-ne un

Finalment es va escollir utilitzar Google SketchUp 8 Pro i es va buscar un model 3D d'un esquelet humà en la llibreria de models que ofereix.

7.2.1. Sketch Up

Fig 18. Logotip del software Google SkethUp

Google SketchUp 8 Pro ens ofereix moltes eines per poder tractar el model 3D de la forma que vulguem. A més, també ofereix una àmplia llibreria de models 3D per tal de buscar el que millor ens vingui a l'hora d'escollir-ne un pel nostre sistema de Realitat Augmentada.

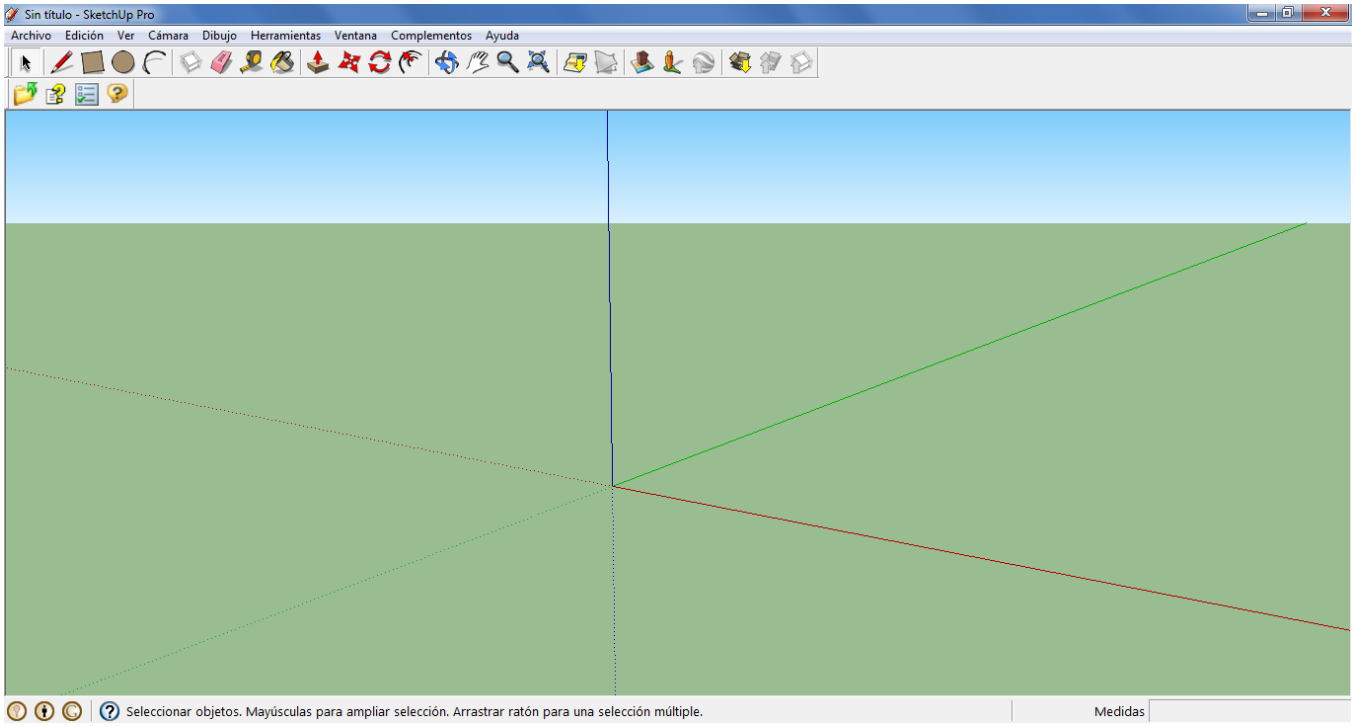


Fig 19. Aspecte que presenta el software Google SketchUp

7.2.2. Model utilitzat

Finalment, es va escollir un model 3D d'un esquelet humà que va semblar prou bo, també es veia de forma bastant clara quins serien els punts per on s'hauria de dividir el model per obtenir les diferents parts del model.

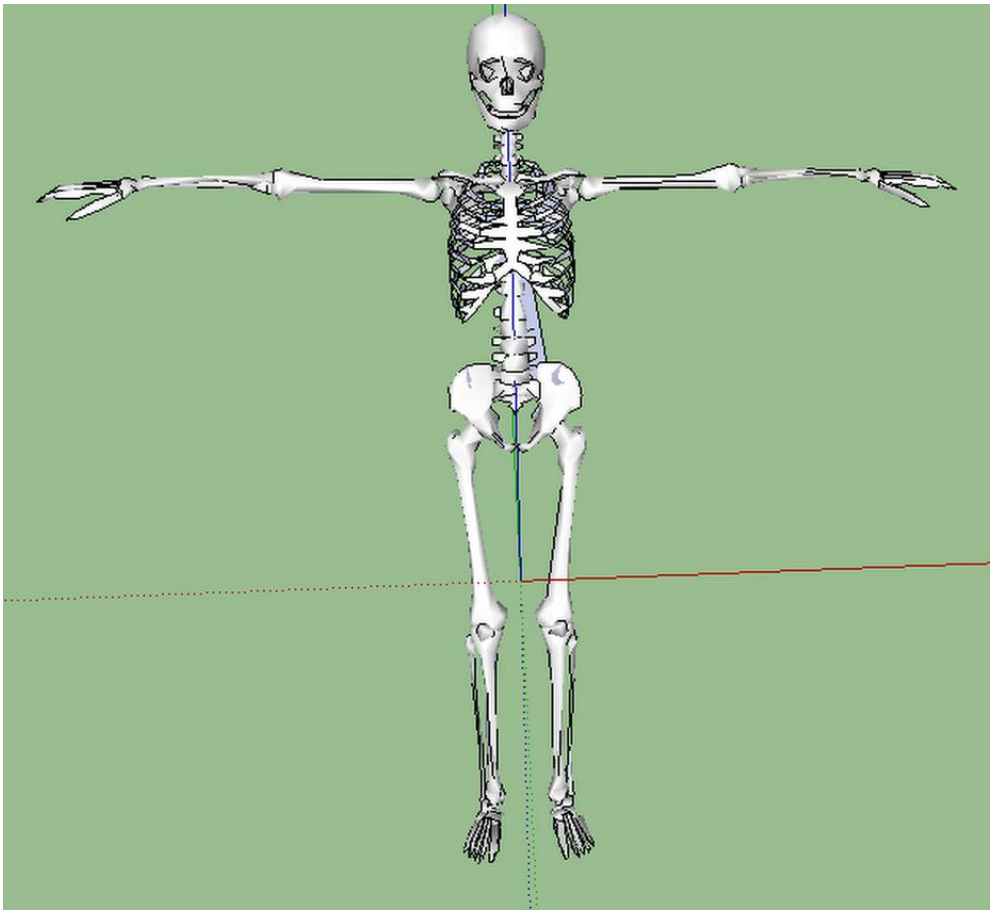


Fig 20. Model 3D utilitzat finalment a la visualització del sistema

7.2.3. Lectura Model

Tal com es parlava al punt 6.1.2. sobre la preparació de l'aplicació pel que fa a models de producció, també cal declarar el model i després fer la crida per tal que es llegeixi el nostre model 3D.

```
Model m;
mod.readModel();
```

Al codi de la classe Object és on es pot trobar el codi que realment s'encarrega de recórrer tot l'arxiu de l'objecte que es vol llegir. Aquest codi fa un recorregut línea per línea, ja que en els fitxers de tipus .obj s'introdueix un vèrtex per cada línea. D'aquesta manera seqüencial es va llegint tot el fitxer i tan bon punt es té disponible un nou vèrtex o una nova cara s'introdueixen a l'estructura de dades on s'han de emmagatzemar.

7.3. Partició del model

Per tal de dividir el model i poder-lo articular posteriorment, hem de tenir presents quines són les articulacions que ens ofereix l'OpenNI. Les articulacions que utilitzem són les següents:

- XN_SKEL_HEAD: punt corresponent al cap de l'usuari
- XN_SKEL_NECK: punt corresponent al clatell de l'usuari
- XN_SKEL_RIGHT_SHOULDER/XN_SKEL_LEFT_SHOULDER: punts corresponents a les espatlles de l'usuari.
- XN_SKEL_RIGHT_ELBOW/XN_SKEL_LEFT_ELBOW: punts corresponents als colzes de l'usuari.
- XN_SKEL_RIGHT_HAND/XN_SKEL_LEFT_HAND: punts corresponents a les mans de l'usuari.
- XN_SKEL_TORSO: punt corresponent al centre del tronc de l'usuari.
- XN_SKEL_RIGHT_HIP/XN_SKEL_LEFT_HIP: punts corresponents als malucs de l'usuari.
- XN_SKEL_RIGHT_KNEE/XN_SKEL_LEFT_KNEE: punts corresponents als genolls de l'usuari.
- XN_SKEL_RIGHT_FOOT/XN_SKEL_LEFT_FOOT: punts corresponents als peus de l'usuari.

7.3.1. Parts resultants

La primera de les parts resultants de dividir el model 3D és la que queda continguda entre el cap (XN_SKEL_HEAD) i el clatell (XN_SKEL_NECK).

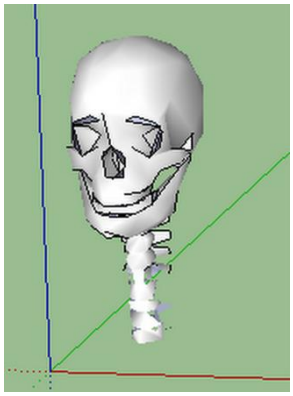


Fig 21. Cap del model

Una altra part de les resultants és la corresponent al tronc del model. Aquesta és la més difícil de tractar tan en la divisió del model com en la seva visualització, donat que intervenen quatre articulacions diferents: les espatlles (XN_SKEL_RIGHT_SHOULDER i XN_SKEL_LEFT_SHOULDER) i els malucs (XN_SKEL_RIGHT_HIP i XN_SKEL_LEFT_HIP).

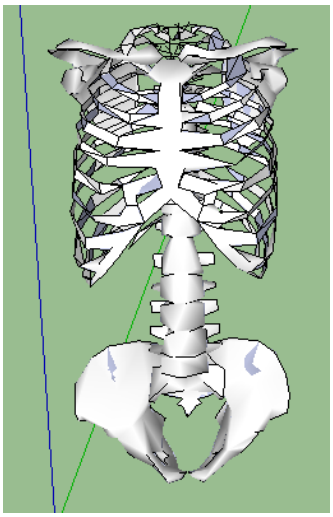
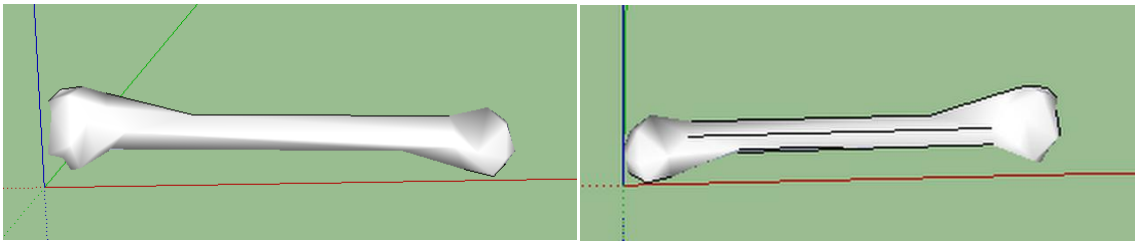


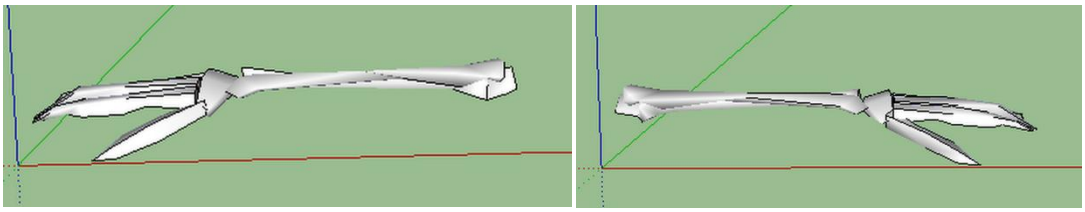
Fig 22. Tronc del model

Les següents dues parts resultants que apareixen són les corresponents als avantbraços, és a dir, les contingudes entre les espatlles (XN_SKEL_RIGHT_SHOULDER i XN_SKEL_LEFT_SHOULDER) i els colzes (XN_SKEL_RIGHT_ELBOW i XN_SKEL_LEFT_ELBOW).



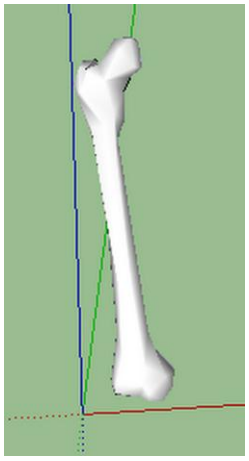
Figs 23 i 24. Avantbraços del model

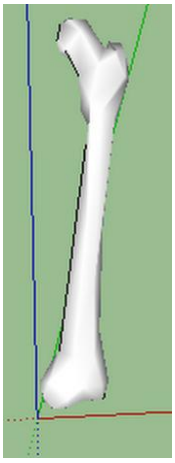
Els braços del model són dues parts que queden a les zones contingudes entre els colzes (XN_SKEL_RIGHT_ELBOW i XN_SKEL_LEFT_ELBOW) i les mans (XN_SKEL_RIGHT_HAND i XN_SKEL_LEFT_HAND).



Figs 25 i 26. Braços del model

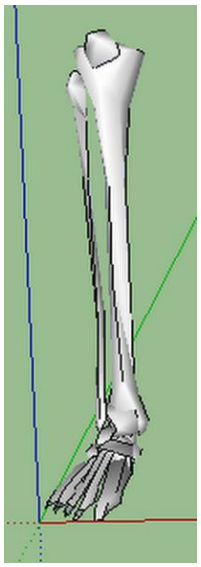
Pel que fa a les cames, trobem les cuixes entre els malucs (XN_SKEL_RIGHT_HIP i XN_SKEL_LEFT_HIP) i els genolls (XN_SKEL_RIGHT_KNEE i XN_SKEL_LEFT_KNEE).

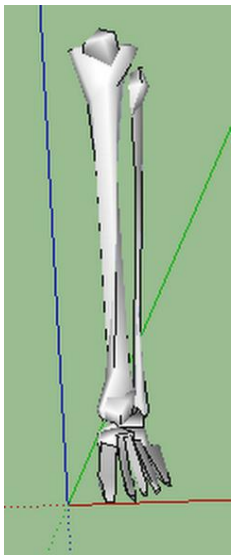




Figs 27 i 28. Cuixes del model

Per últim, les cames del model queden a les zones entre els genolls (XN_SKEL_RIGHT_KNEE i XN_SKEL_LEFT_KNEE) i els peus (XN_SKEL_RIGHT_FOOT i XN_SKEL_LEFT_FOOT).





Figs 29 i 30. Cames del model

D'aquesta manera obtenim les 10 diferent parts que s'hauran de llegir i emmagatzemar per separat per tal de tenir diferents objectes que siguem capaços d'articular mitjançant les transformacions geomètriques que siguin necessàries.

El codi encarregat de llegir totes les parts resultants de la divisió és el que es troba a la funció Model::readModel():

```
string path (string part)
{
    string res = "";//Path absolut del directori
    res += "model_" + part + ".obj";
    return res;
}

void Model::readModel()
{
    cabeza.readObj(path("cabeza"),matlib);
    tronco.readObj(path("tronco"),matlib);
    antebrazo_izqdo.readObj(path("antebrazo_izqdo"),matlib);
    antebrazo_drcho.readObj(path("antebrazo_drcho"),matlib);
    brazo_izqdo.readObj(path("brazo_izqdo"),matlib);
    brazo_drcho.readObj(path("brazo_drcho"),matlib);
    antepierna_izqda.readObj(path("antepierna_izqda"),matlib);
    antepierna_drcha.readObj(path("antepierna_drcha"),matlib);
    pierna_izqda.readObj(path("pierna_izqda"),matlib);
    pierna_drcha.readObj(path("pierna_drcha"),matlib);
}
```

8. Visualització i resultats

En aquest apartat s'acaba d'explicar detalladament la part de l'aplicació que s'ha desenvolupat per tal de suportar el bloc de Visualització del sistema de Realitat Augmentada.

Fig 31. Blocs necessaris per construir un sistema de Realitat Augmentada

Concretament, en aquest apartat s'explica tots els resultats que s'han obtingut al llarg del desenvolupament de l'aplicació. També s'explica amb detall els mètodes utilitzats per visualitzar l'escena de la forma que es pretén. Finalment, també s'explica les transformacions geomètriques que s'apliquen a cada objecte resultant de dividir el model, per tal de veure el model 3D articulat.

8.1. Resultats parcials

L'aplicació es comença tenint com a base un dels programes d'exemple que ens ofereix OpenNI una vegada el tenim instal·lat. El que ens mostra aquest programa d'exemple és una imatge de profunditats. Quan detecta que un nou usuari ha entrat a l'escena, el pinta d'un color. Poden entrar un o més usuaris a l'escena i li assigna un color diferent a cadascun, de manera que queda prou diferenciat quines parts de l'escena ocupa cada usuari.

Quan un usuari agafa la posició concreta, gràcies a la capacitat del node de producció de seguiment dels usuaris Pose Detection, es dibuixa l'esquelet dels usuaris dels que s'ha detectat aquesta posició. L'esquelet que es pinta consisteix en línies entre les diferents articulacions a les que ens permet accedir l'OpenNI.

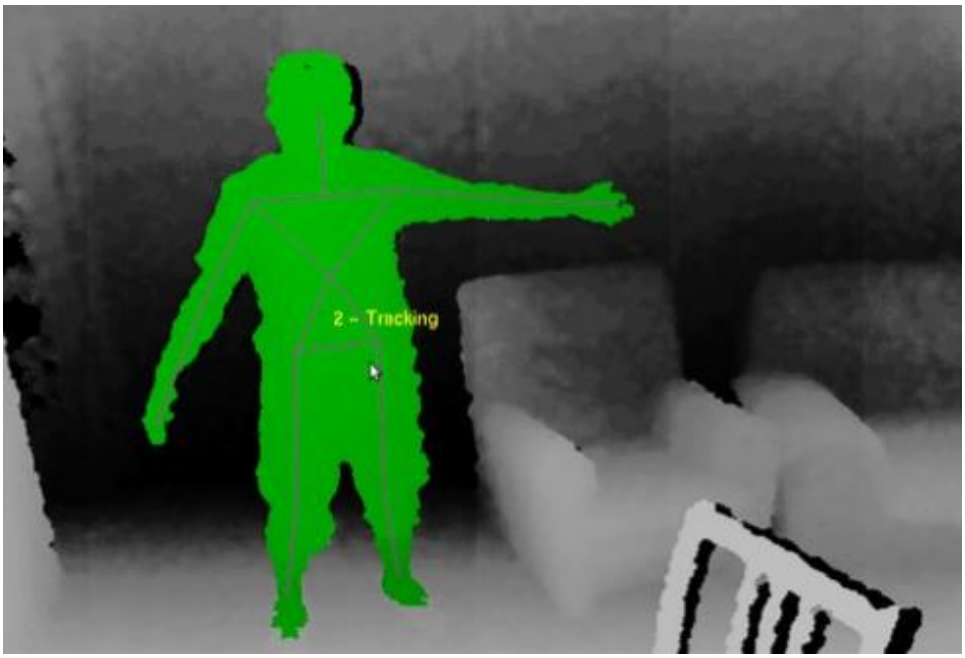


Fig 32. Primer resultat parcial obtingut

El següent pas que es va portar a terme va ser visualitzar la imatge real que rep la càmera (mapes de imatges a color) en comptes de les imatges de profunditat com a imatge de fons a la visualització. D'aquesta manera s'aconsegueix que l'usuari es vegi a sí mateix i el seu entorn, però una vegada el sistema el detecta comença a veure elements addicionals.

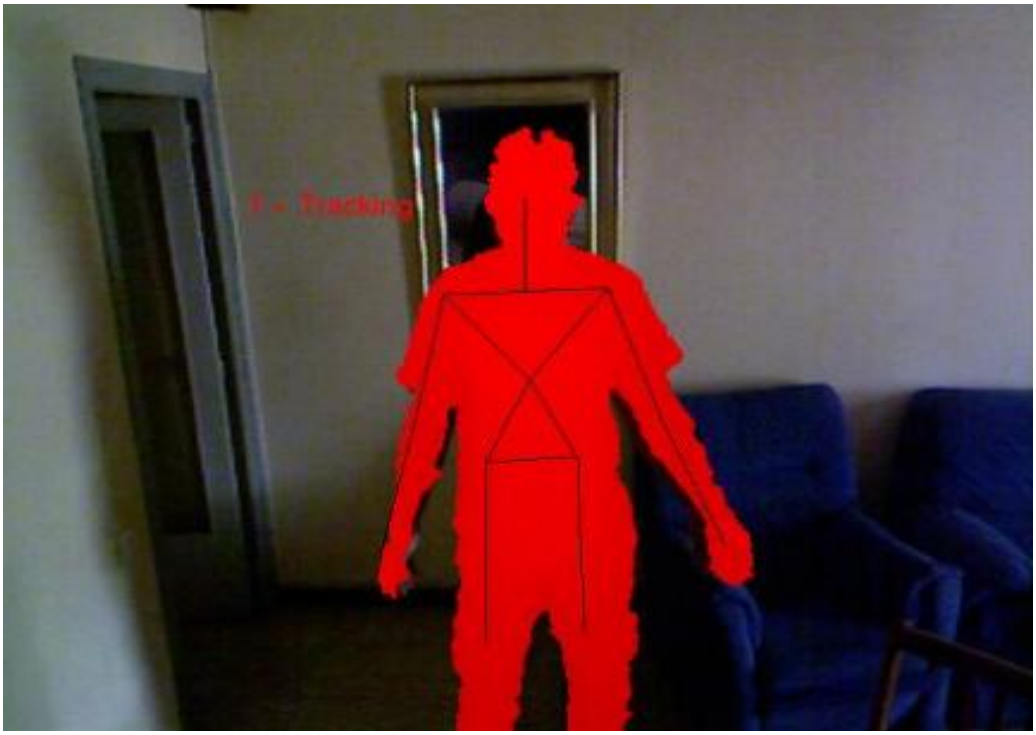


Fig 33. Segon resultat parcial obtingut

A continuació el que es va fer fou que no aparegués l'usuari en cap moment. Es fa que al començament de l'execució es carregui el fons de l'escena sense cap usuari al davant. En tot moment es mostra aquesta imatge com a imatge de fons.

Una vegada l'usuari es posiciona davant del sensor i agafa la posició necessària per tenir enlloc la posició de les seves articulacions, es dibuixa únicament l'esquelet dibuixat amb línies que havíem vist anteriorment.

Com a resultat, l'usuari visualitza l'escena buida tot i que ell n'estigui a l'escena. Una vegada agafa la posició però, veu un esquelet dibuixat amb línies movent-se exactament a com s'està movent ell.

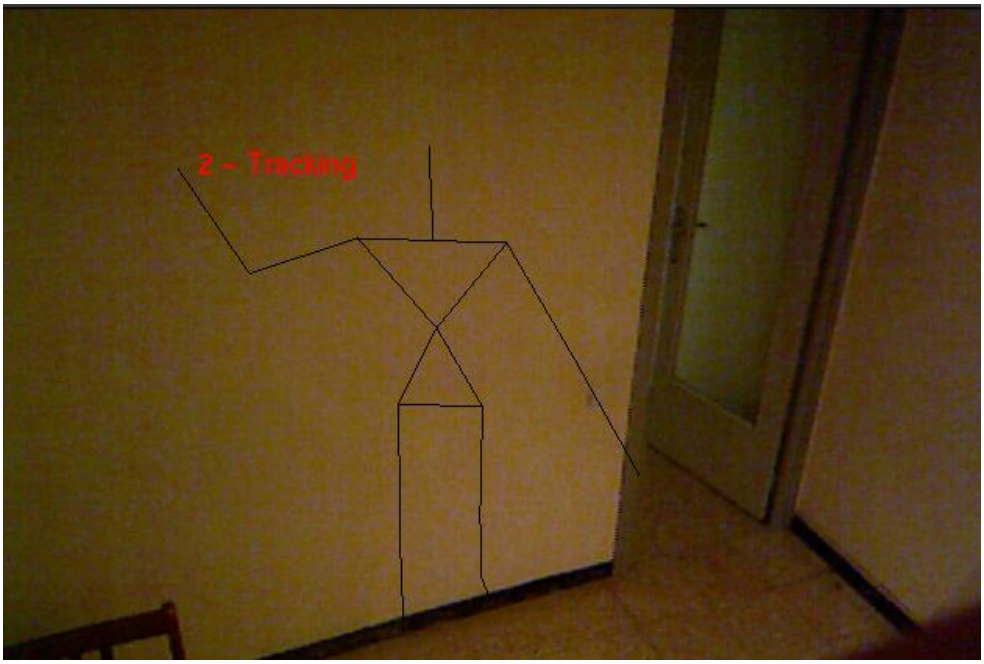


Fig 34. Tercer resultat parcial obtingut

Una vegada s'arriba a aquest punt, ens proposem canviar la forma de visualitzar l'esquelet. En comptes de dibuixar línies entre les articulacions volem pintar cadascuna de les parts del model 3D que hem dividit anteriorment. Per tal d'aconseguir aquest objectiu, s'ha d'entrar amb més profunditat en el funcionament de OpenGL i les opcions que posa a la nostra disposició.

8.2. OpenGL

El primer tractament que es fa amb OpenGL és al mateix punt on es prepara el controlador OpenNI per tal de començar a generar dades.

Per tal de preparar l'OpenGL utilitzem el següent codi:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutInitWindowSize(GL_WIN_SIZE_X/2, GL_WIN_SIZE_Y/2);
glutCreateWindow ("Sistema de Realitat Augmentada amb Kinect");
glutSetCursor(GLUT_CURSOR_INHERIT);

glutKeyboardFunc(glutKeyboard);
glutDisplayFunc(glutDisplay);
glutIdleFunc(glutIdle);
```

```
glDisable(GL_DEPTH_TEST);
glEnable(GL_TEXTURE_2D);

glutMainLoop();
```

Després de cridar les funcions d'inicialització per l'aplicació i la finestra que volem que es mostri, es crea una finestra amb el títol que es vol mostrar ("Sistema de Realitat Augmentada amb Kinect").

Es vinculen les funcions que atendran les interrupcions, glutKeyboard() s'executarà quan es polsi una tecla del teclat i glutDisplay() serà la funció que s'executarà per cada frame. Finalment es crida a glutMainLoop() per tal de començar a processar frames i que es comenci a executar la part principal de l'aplicació.

Hi ha dos formes de visualització depenent de quin punt dels resultats previs ens hi trobem. En els primers passos que es van fer, es una imatge de profunditats o una imatge a color i després es dibuixava algun element al damunt. En els últims passos, es dibuixa sempre el mateix com a imatge de fons i es superposa algun element addicional.

En tots els casos, el primer que s'ha de fer és situar el viewpoint de manera que s'observi la regió de l'escena que pintarem a continuació. El codi utilitzat per això és:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glOrtho(0, GL_WIN_SIZE_X, GL_WIN_SIZE_Y, 0, -1.0, 1.0);
```

Entrem en mode de projecció i es configura la càmera ortogonal que observarà l'escena que volem pintar. Es fa de manera que puguem pintar tota la imatge sense haver de controlar límits de la finestra. Una vegada ja està fet aquest pas, podem dibuixar allò que vulguem a l'escena que s'està observant. En funció de si volem pintar la imatge emmagatzemada al principi de l'execució o la imatge real que capta la càmera, s'executa una crida o una altra a la funció glTexImage2D(). El codi utilitzat és:

```
if (g_nViewState >= DISPLAY_MODE_NO_USER)
{
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, g_nTexMapX,
		g_nTexMapY, 0, GL_RGB, GL_UNSIGNED_BYTE, im);
}
else {
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, g_nTexMapX,
		g_nTexMapY, 0, GL_RGB, GL_UNSIGNED_BYTE, g_pTexMap);
}
```

Es pot veure que si tenim un mode determinat (DISPLAY_MODE_NO_USER o posterior, que són els que pinten com a imatge de fons una fixa) es dibuixa el que tenim emmagatzemat i en cas contrari es passa coma paràmetre un altre conjunt de dades, que és el que s'ha estat tractant.

Finalment, depenent de si s'ha de dibuixar l'esquelet format per línies o el model 3D, es fa una cosa o una altra. Si s'ha de dibuixar l'esquelet format per línies es crida la funció DrawLimb(), que s'ocupa de dibuixar una línia entre els dos punts que corresponen a les articulacions que se li passen com a paràmetre.

Si, per una altra banda, el que es vol pintar és el model 3D, es redefineix la càmera ortogonal que observa l'escena per tal de tenir unes altres mesures a l'hora de pintar. El que es fa és declara el View Reference Point (VRP), la posició on ha d'estar la font de llum que volem que il·lumini el model i definim la càmera ortogonal amb unes dimensions que ens faran més amigables el procés de pintat del model.

```
GLfloat point[] = {0.0, 0.0, 0.5, 1.0};
GLfloat ambient[] = {1.0, 1.0, 1.0, 1.0};
glLightfv(GL_LIGHT0,GL_POSITION, point);
glLightfv(GL_LIGHT0,GL_AMBIENT, ambient);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0,1.0,-1.0,1.0,-1.0,10.0);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(-point[0],-point[1],-point[2]);
```

A continuació, per cada objecte dels resultants de la divisió del model en parts, s'apliquen les transformacions geomètriques necessàries. Se'n parlarà amb més detall de cada una d'aquestes transformacions a l'últim apartat d'aquest capítol.

Per cadascun dels objectes, es crida a la funció Object::Render(), aquesta funció s'encarrega de dibuixar l'objecte en qüestió.

```
void Object::Render()
{
  Color col;
  Vector vec;
  Model mod;
  Material m = mod.matlib.material(0);
  for (unsigned int i = 0; i < faces.size(); i++) {
    vec = faces[i].normal;
    col = m.ka;
    GLfloat aa[4] = {col.r,col.g,col.b,col.a};
    glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,aa);
    col = m.kd;
    GLfloat ab[4] = {col.r,col.g,col.b,col.a};
    glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,ab);
    col = m.ks;
    GLfloat ac[4] = {col.r,col.g,col.b,col.a};
    glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,ac);
    glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,m.shininess);
    glBegin(GL_POLYGON);
    glShadeModel(GL_SMOOTH);
    for (unsigned int j = 0; j < faces[i].vertices.size(); j++) {
       int id = faces[i].vertices[j];
       glNormal3f(vertices[id].normal.x,vertices[id].normal.y,
                  vertices[id].normal.z);
```

```
            glVertex3f(vertices[id].coord.x,vertices[id].coord.y,
                       vertices[id].coord.z);
        }
        glEnd();
      }
    }
```

Es pot veure com, per cada cara de l'objecte, es configuren les propietats de color del material del que està formada la cara. Amb la funció glMaterialfv() es configuren les llums ambient, difosa i especular, així com la brillantor de la cara. Finalment es pinten els tres vèrtexs de cada cara, passant informació de les normals de cada vèrtex.

8.3. Transformacions del model

Tots els objectes resultats de la divisió del model inicial són tractats amb transformacions geomètriques de manera que apareguin allà on nosaltres volem.

Per tots els objectes, la primera transformació que se li aplica és moure tot l'objecte al centre de coordenades. Per tal de fer aquesta translació utilitzem les mides de la capsa contenidora de l'objecte.

El següent pas és fer les rotacions necessàries per pintar l'objecte amb les orientacions que nosaltres volem. Per això es tenen en compte les posicions de les articulacions que delimiten cadascuna de les parts de l'objecte i es fan els càlculs necessaris per calcular l'angle de rotació respecte a cadascun dels eixos.

A continuació s'ha de fer l'escalat per tal d'aconseguir que els objectes que estem pintant tinguin una mida adequada. Per aconseguir que quedi amb la mida exacta, ni més gran ni més petit del que hauria, aprofitem les mides de la capsa contenidora i la diferència de coordenades entre els punts de les articulacions. La relació que volem per fer l'escalat és la diferència de coordenades dividit per les mides de la capsa contenidora.

Com a últim pas s'ha de transportar cadascun dels objectes a la posició que finalment han d'ocupar a l'escena que es vol visualitzar. Per aconseguir això, fixem per cada objecte l'articulació que ens determinarà el punt on s'ha de moure tot l'objecte.

Com a resultat final de la visualització, obtenim l'esquelet dibuixat on hauria de ser l'usuari. Per tant, l'usuari es troba una mena de mirall on veu l'entorn que l'envolta però no es veu a sí mateix, sinó un esquelet que es mou tal i com ho fa ell.

9. Conclusions

Es volia aconseguir un sistema de Realitat Augmentada en què els usuaris poguessin interactuar amb el seu entorn o bé veure'l modificat.

Finalment s'ha aconseguit un sistema de Realitat Augmentada en què els usuaris poden veure un esquelet en 3D movent-se tal i com ho fan ells, és a dir, poden veure l'escena a la que ells es troben modificada, ja que ells no apareixen.

Han influït bona part dels coneixements obtinguts al llarg dels estudis a la facultat. S'han tingut en compte al llarg del projecte coneixements de sistemes operatius, per tal d'instal·lar sense problemes tot el software necessari i les seves llibreries. Han contribuït els coneixements en visió per computador a l'hora d'entendre el funcionament del dispositiu Microsoft Kinect, així com el tractament que en fa de les dades que capta. També s'han aprofitat coneixements d'enginyeria del software a l'hora de dissenyar una estructura de dades de manera que el disseny resultant sigui canviable i transparent a l'aplicació que interactua amb aquesta estructura de dades. Finalment, han sigut molt útils els coneixements de visualització gràfica, ja que han sigut decisius per fer avançar el projecte en la direcció adequada quan havia problemes de visualització.

El resultat del projecte és molt vistós i això ajuda força quan se'l troba un usuari de fora del projecte. S'ha aconseguit un bon resultat tot i les complicacions que van sorgir a mesura que avançava el projecte.

10. Pressupost i planificació

En aquest apartat es pretén donar un pressupost del projecte al complet i la planificació que s'ha seguit al llarg del projecte.

Els elements necessaris per portar a terme el projecte són:

- Ordinador portàtil amb sistemes operatius Windows i Linux
- Microsoft Kinect
- Google SketchUp 8 Pro

L'ordinador utilitzat està valorat en 499€, suposant que es compra amb Windows instal·lat de fàbrica i s'ha de pagar la llicència. El dispositiu Microsoft Kinect es troba actualment valorat en 149.99€. La llicència corresponent a la versió professional del software Google SketchUp 8 té un preu de $495. Si calculem el preu d'aquesta llicència en euros, tenim un preu de 388.77€.

El preu dels elements utilitzats puja fins a 499€ + 149.99€ + 388.77€ = 1037.76€

Pel que fa a les hores invertides en el projecte, s'han invertit un total de 380 hores. Tenint en compte el sou d'un treballador de 20€/hora. El cost del projecte en termes de sou de treballadors suposa 380hores*20€/hora = 7600€.

El preu total del producte és la suma dels costos del sou de treballador i el preu dels elements utilitzats:

7600€ + 1037.76€ = **8637.76€**

Concepte	Quantitat	Preu	Preu Total
Ordinador portàtil	1	499 €	499 €
Microsoft Kinect	1	149.99 €	149.99 €
Google SketchUp 8 Pro	1	388.77 €	388.77 €
Hores dedicades	380	20 €	7600 €
Total			8637.76 €

Fig 35. Càlcul del pressupost

La planificació es mostra a la pàgina següent en horitzontal de manera que es pugui veure de forma més clara.

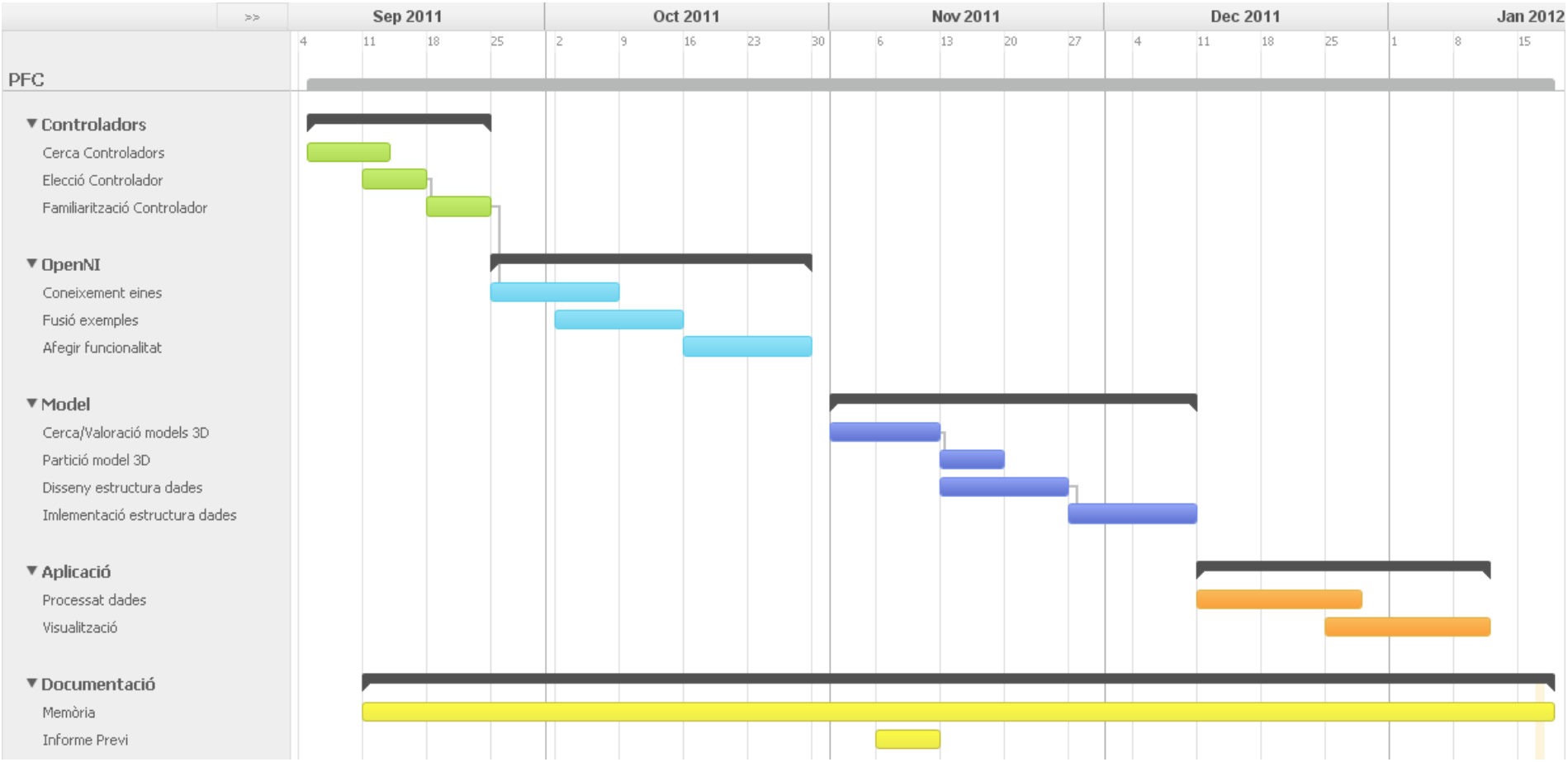


Fig 36. Planificació del projecte

11. Glossari de termes

Articulació: Unió entre dos parts del cos de l'usuari.

Callback: Devolució d'una crida d'una funció. Fa referència a quan es prepara una o més funcions per què s'executi/n quan passa un event determinat.

Controlador: Conjunt de software que s'encarrega d'accedir a les dades capturades pel sensor i retornar-ne els resultats de la captura.

Imatge de color: Imatge real que rep la càmera del dispositiu, és el tipus d'imatge que rep l'ull humà.

Imatge de profunditat: Imatge creada de forma artificial on cada píxel té una intensitat diferent en funció de si el punt corresponent a aquest píxel es troba a poca o a molta distància del sensor.

Middleware: Software que assisteix a una aplicació per interactuar amb altres aplicacions o també amb sistemes operatius, xarxes o elements hardware.

Model: Estructura o conjunt de dades que representa quelcom del món real. En el nostre cas, el model sempre representa un cos humà.

Món real: Entorn físic que envolta els usuaris que interactuen amb el sistema i que es pot observar a través dels sensors del dispositiu.

Món virtual: Conjunt d'elements virtuals que crea l'aplicació. Normalment és un conjunt de dades que es representa d'alguna forma determinada.

Usuari: Qualsevol persona que es situa davant del sensor per tal d'utilitzar el sistema de Realitat Augmentada.

12. Bibliografia

[1] http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/
[2] http://www.openni.org/
[3] http://remendozag.blogspot.com/2011/02/como-muchos-ya-saben-han-salido-la-luz.html
http://www.xbox.com/es-es/kinect
http://www.wikipedia.com/