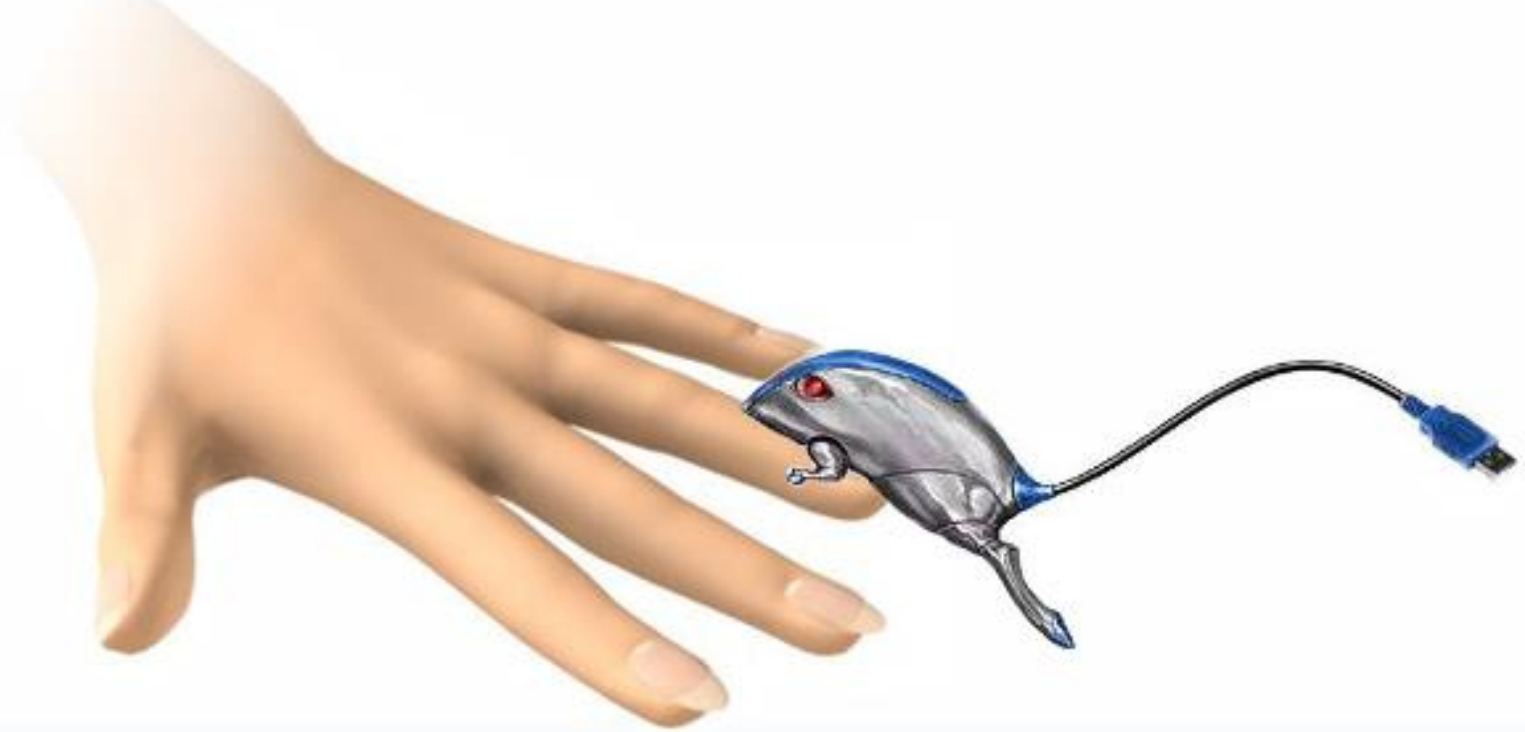
“Hands-On” 5DT Data Glove



gesticolando tra C/C++ e Quest3D

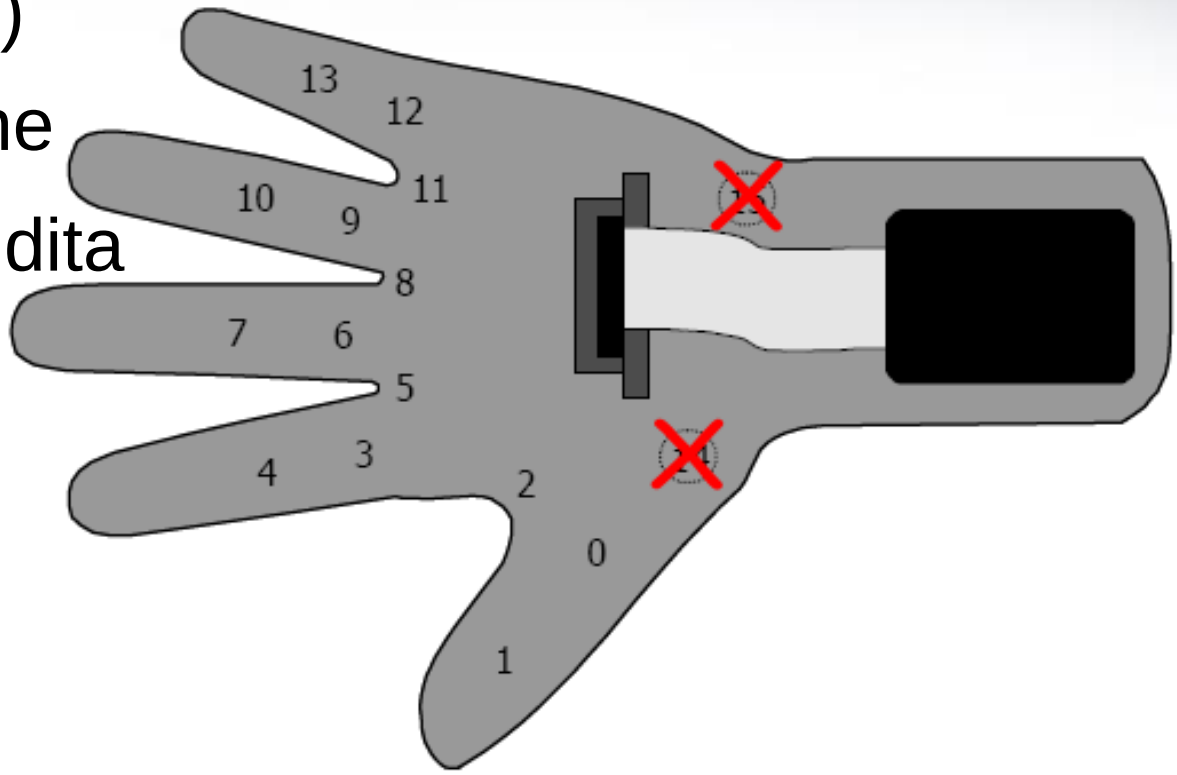
Dario Scarpa
darioscarpa@duskzone.it

L'hardware

- 5DT Data Glove 16
  - Guanto in lycra che ospita dei sensori realizzati con filamenti in fibra ottica
  - Un'unità optoelettronica e una piccola interface box provvedono a inviare i valori rilevati dai sensori sulla porta seriale alla quale si collega il guanto

Sensori... 14 o 16?

- 10 valori di flessione
  - due per dito (alla nocca e tra falange e falangina)
- 4 valori di divaricazione
  - (tra ogni coppia di dita adiacenti)
- E i sensori 14/15?



I sensori fantasma

- I due sensori previsti dallo schema ma marcati come “non ancora implementati” sono
  - Thumb translation
  - Wrist flexure
- Peccato!
  - la flessione del polso avrebbe potuto fornire una prima indicazione del movimento della mano nello spazio, e non solo delle dita rispetto alla mano...
  - la traslazione del pollice avrebbe permesso di modellare con precisione l'”opponibilità del pollice” :)

Curiosità: il protocollo di comunicazione (1/3)

- Il guanto e' un dispositivo transmit-only
- usa solo le linee TX/GND della RS-232
- Il data stream inviato dal guanto consiste di due tipi di pacchetti, ***gdata16*** e ***ginfo16***
  - ***gdata16***
    - ***36 bytes:***
      - ***0-1 | header: “<D”***
      - ***2-33| 2 bytes s_high e s_low per ogni sensore:***
        - ***sensorvalue = (s_high) x 256 + s_low***
      - ***34 | Checksum (byte meno significativo della somma di tutti i valori dei sensori nel pacchetto)***
      - ***35 | trailer: “>”***

Curiosità: il protocollo di comunicazione (2/3)

- ***ginfo16***
  - ***7 bytes:***
    - ***0-1 | header: “<I”***
    - ***2 | d1: version major***
    - ***3 | d2: version minor***
    - ***4 | d3: capability word low***
      - ***Bit 0 settato se guanto mano destra***
      - ***Bit 6 settato se guanto mano sinistra***
    - ***5 | d4: capability word high***
      - ***Bit 0 settato se versione wireless del guanto***
    - ***6 | trailer: “>”***

Curiosità: il protocollo di comunicazione (3/3)

Come é facile immaginare...

- ***Un flusso continuo di pacchetti gdata16 comunica al driver i valori rilevati dai sensori***
- ***La trasmissione occasionale di pacchetti ginfo16 permette invece di determinate alcune caratteristiche del guanto in uso***

```
... gdata16 gdata16 gdata16 ginfo16 gdata16 gdata16 ...
```

Il Driver

- Per utilizzare il guanto, si può tranquillamente ignorare il protocollo seriale e ricorrere al driver fornito
  - Windows (fglove.dll + fglove.lib e fglove.h)
  - Linux (libfglove.so + fglove.h)
- un'API cross-platform fornita dal driver permette di aprire/chiudere il device, gestire la calibrazione dei sensori e ottenere i valori correnti dei sensori dal guanto, nonché (molto limitatamente) il matching con un set di 16 “gesture” predefinite

Analizziamo l'API un po' più in dettaglio...

L'API

- Le funzioni messe a disposizioni dall'API si possono cosi' raggruppare:
  - Accesso al dispositivo e informazioni sullo stato
  - Gestione della calibrazione e dello scaling dei valori
  - Accesso ai valori dei sensori
    - raw
    - scaled
  - Rilevamento “gestures”

L'API in dettaglio: funzioni poco interessanti

- **fdGlove *fdOpen(char *pPort);**
  - Indicata la porta seriale a cui e' connesso il guanto, restituisce un puntatore alla struttura fdGlove che lo rappresenta
- **int fdClose(fdGlove *pFG);**
  - Rilascia la risorsa guanto

L'API in dettaglio: funzioni poco interessanti

- **int fdGetGloveHand(fdGlove *pFG);**
  - Guanto destro o sinistro? (***FD_HAND_LEFT* o *FD_HAND_RIGHT*)**
- **int fdGetGloveType(fdGlove *pFG);**
  - Che tipo di guanto é connesso, se é connesso? (***FD_GLOVENONE*, *FD_GLOVE7*, *FD_GLOVE7W*, *FD_GLOVE16*, *FD_GLOVE16W*)**
- **int fdGetNumSensors(fdGlove *pFG);**
  - Il numero di valori dei sensori che il driver rende disponibile
    - Attenzione: non per forza il numero dei sensori realmente presenti sul guanto
    - Correntemente, comunque, sembrerebbe hardcoded a 18

L'API in dettaglio: funzioni poco interessanti

- **void fdGetGloveInfo(fdGlove *pFG, unsigned char *pData);**
  - Ottiene l'"information data block" del guanto connesso (32 bytes)
- **void fdGetDriverInfo(fdGlove *pFG, unsigned char *pData);**
  - Ottiene l'"information data block" del driver (32 bytes, NULL terminated string)

L'API in dettaglio: i valori “raw” dei sensori

- I valori raw dei sensori si prelevano con
  - void fdGetSensorRawAll(fdGlove *pFG, unsigned short *pData);
  - unsigned short fdGetSensorRaw(fdGlove *pFG, int nSensor);
- Il valore “raw”di un sensore é un intero senza segno di 12 bit (0-4095)
  - Su questi si può basare una routine di scaling personalizzata (ci torneremo parlando della calibrazione...)
- Le funzioni dell'API che hanno a che fare con tali valori raw lavorano quindi con degli ***unsigned short***

L'API in dettaglio: i valori “raw” dei sensori

- ***Molte delle funzioni dell'API hanno due versioni: una per interagire con un singolo sensore e la corrispettiva “All” per interagire con tutti...***
- Recuperare i valori di tutti i sensori in una sola chiamata

```
unsigned short *sensorValues;

sensorValues = (unsigned short *)
    malloc(fdGetNumSensors(glove)*sizeof(unsigned short));

fdGetSensorRawAll(glove, sensorValues);
```

- Recuperare il valore di uno specifico sensore

```
unsigned short indexKnuckleFlex = fdGetSensorRaw(glove, FD_INDEXNEAR);
```

L'API in dettaglio: forzare i valori raw

E' possibile anche “forzare” dei valori nel buffer dei valori grezzi gestito dal driver

- void fdSetSensorRawAll(fdGlove *pFG, unsigned short *pData);
- void fdSetSensorRaw(fdGlove *pFG, int nSensor, unsigned short nRaw);
  - *Ma a che scopo?*

L'API in dettaglio: i valori scalati

Oltre che i valori “raw”, si possono recuperare dal guanto dei **valori scalati**, ovvero dei float ottenuti dai valori raw in base all'autocalibrazione effettuata dal guanto *(descritta tra qualche slide)*

- void fdGetSensorScaledAll(fdGlove *pFG, float *pData);
- float fdGetSensorScaled(fdGlove *pFG, int nSensor);

L'API in dettaglio: i valori scalati

- Di default il range dei valori scalati é [**0**..**1**], ma volendo lo si può alterare
- Get/Set del valore massimo del range
  - void fdGetSensorMaxAll(fdGlove *pFG, float *pMax);
  - float fdGetSensorMax(fdGlove *pFG, int nSensor);
  - void fdSetSensorMaxAll(fdGlove *pFG, float *pMax);
  - void fdSetSensorMax(fdGlove *pFG, int nSensor, float fMax);

L'API in dettaglio: la calibrazione

Definiamo come **Dynamic Range** di un sensore la differenza del valore raw del sensore con mano completamente chiusa e completamente aperta:

DynamicRange = ValueMax – ValueMin

- Con mani diverse si hanno DynamicRange diversi
  - E' quindi necessaria una calibrazione software per **normalizzare** i valori indipendemente dai diversi DynamicRange

L'API in dettaglio: la calibrazione

- Teniamo traccia del **massimo** e del **minimo** dei valori rilevati per un certo sensore (col guanto indossato da un utente X), e poi scaliamo i valori rilevati al range massimo desiderato (es: 255)

ValueScaled = (ValueMeasured – ValueMin)*(255/DynamicRange)

L'API in dettaglio: la calibrazione

- Una calibrazione del guanto, concretamente, non é che la definizione del range rilevato (con una certa mano) per ognuno dei sensori.
  - Ovvero, due array di unsigned short contenente i valori raw massimi e minimi necessari a calcolare i valori scalati...

L'API in dettaglio: l'autocalibrazione

- Il driver implementa una routine di autocalibrazione dinamica
- A partire dall'inizializzazione del guanto, per ogni sensore, i valori letti vengono continuamente confrontati con i valori massimo e minimo rilevati (aggiornandoli quando opportuno)
  - Chiamiamo tali valori RawMin e RawMax
    - Sono i valori che si possono impostare tramite le funzioni

```
fdSetCalibrationAll()
fdSetCalibration()
fdResetCalibration().
```

$$out = \frac{(RawVal - RawMin)}{(RawMax - RawMin)} Max$$

  - *È il valore scalato ottenuto, che appartiene al range [0...Max] (di default, [0..1])*

L'API in dettaglio: l'autocalibrazione

- *In pratica, per calibrare il guanto, basta aprire/chiudere la mano qualche volta per far si' che vengano settati dei valori appropriati per RawMin e RawMax*
- *Volendo, si può ignorare del tutto l'autocalibrazione e basare una propria routine di scaling sui valori raw*

L'API in dettaglio: gestire l'autocalibrazione

- La (auto)calibrazione attiva si manipola con due coppie di funzioni Get/Set
  - Come prevedibile, si passano/prelevano i valori RawMax/RawMin per uno o per tutti i sensori
    - Avendo a che fare con i valori raw, i parametri sono unsigned short ...

```
void  fdGetCalibrationAll(fdGlove *pFG, unsigned short *pUpper, unsigned short *pLower);
void  fdGetCalibration(fdGlove *pFG, int nSensor, unsigned short *pUpper, unsigned short *pLower);
void  fdSetCalibrationAll(fdGlove *pFG, unsigned short *pUpper, unsigned short *pLower);
void  fdSetCalibration(fdGlove *pFG, int nSensor, unsigned short nUpper, unsigned short nLower);
```

L'API in dettaglio: gestire l'autocalibrazione

- Reset della calibrazione

  void fdResetCalibration(fdGlove *pFG);

  - Resetta la calibrazione del guanto a valori appropriati
  - Equivale a chiamare fdSetCalibrationAll con valori massimi a 0 e valori minimi a 4095
    - Per i sensori non presenti in hardware viene fatto il contrario: massimo 4095, minimo 0

L'API in dettaglio: ...calibrazione...?

- Riassumendo, si può scegliere di
  - Ignorare completamente la gestione della calibrazione: affidarsi all'autocalibrazione e prelevare i float scalati, nel range [0..1]
    - Pro
      - non si deve fare nulla...
    - Contro
      - appena inizializzato il guanto, si hanno valori totalmente sballati
      - l'utente e' costretto all'apri/chiudi/apri mano all'inizio di ogni utilizzo

L'API in dettaglio: ...calibrazione...?

- Ignorare completamente la gestione di scaling e calibrazione effettuate dal driver e prendere soltanto i valori “raw”, provvedendo autonomamente alla normalizzazione dei valori
  - Pro
    - Controllo totale di calibrazione/scaling
  - Contro
    - Più codice (anche se banale) e alla fin fine, é da verificare cosa si possa ottenere concretamente in più gestendo autonomamente scaling e calibrazione...
  - ***Oppure...***

L'API in dettaglio: ...calibrazione...?

- Un'alternativa ragionevole?
  - Usare la gestione della calibrazione e lo scaling del driver
    - ma salvare e caricare le calibrazioni per ogni utente, in modo da evitare il comportamento anomalo del guanto “appena inizializzato”, e aggirando la necessità di dover calibrare il guanto ogni volta

L'API in dettaglio: riconoscimento gesture

Esaminiamo rapidamente l'ultimo “gruppo” di funzioni dell'API...

- **int fdGetNumGestures(fdGlove *pFG);**
  - Il numero di gesti che il driver può riconoscere: nell'implementazione attuale, è 16
- **int fdGetGesture(fdGlove *pFG);**
  - L'id (0-15) del gesto riconosciuto, o -1 se non risconosciuto alcun gesto

L'API in dettaglio: riconoscimento gesture

- Sistema molto grezzo
  - ogni dito viene considerato semplicemente aperto/chiuso, ed il pollice é ignorato: i 16 gesti riconosciuti non sono che le 2^4 combinazioni delle posizioni aperto/chiuso delle 4 dita considerate
- Un dito viene considerato aperto/chiuso in base a un certo valore di **threshold** che é possibile impostare a piacimento...
  - void fdGetThresholdAll(fdGlove *pFG, float *pUpper, float *pLower);
    void fdGetThreshold(fdGlove *pFG, int nSensor, float *pUpper, float *pLower);
    void fdSetThresholdAll(fdGlove *pFG, float *pUpper, float *pLower);
    void fdSetThreshold(fdGlove *pFG, int nSensor, float fUpper, float fLower);

Non vale pero' la pena di scendere nei dettagli...

L'API in dettaglio: i limiti del riconoscimento gesture

- Funzionalità pensata probabilmente per il modello di guanto dotato solo di 5 sensori
  - ma anche in quel caso, perche' non usare 5 bit al posto di 4 e includere così anche le combinazioni col pollice? Mistero...
- E' ragionevole ignorare completamente il riconoscimento gesture: possiamo (con poco sforzo) implementare di meglio.

ISISGloveAPI (?)

- Valeva la pena inserire un layer addizionale sull'API low level fornita?
  - Secondo me si'!
    - L'API del driver non é totalmente immediata...
      - Due versioni per molte delle funzioni (per agire su un sensore o tutti)
      - Set di funzioni abbastanza inutili (riconoscimento gesture... cambio del range dei valori scalati)
      - Gestione della calibrazione un po' macchinosa
    - E' tutto sommato un compito poco impegnativo
      - Mi é venuto naturale farlo sperimentando l'API fornita
      - Mi serviva una scusa per iniziare a imparare C++ provenendo da C/Java

ISISGloveAPI (?)

- ISISGloveAPI è
  - un wrapping a oggetti dell'API low level
  - funzionalità aggiuntive più “ad alto livello”
    - Abbiamo semplicemente a che fare con le classi
      - Glove
      - GloveCalibration
      - HandPosition (?)
        - RawHandPosition
        - ScaledHandPosition

ISISGloveAPI - Glove

- Glove
  - Modella la risorsa guanto
- Il costruttore wrappa fdOpen per aprire il guanto e preleva le info immutabili relative al guanto “una volta per tutte”, come il numero di sensori, rese poi disponibili da vari “getters”...

- Glove(void);
- ~Glove(void);
- bool isRightHand();
- bool isLeftHand();
- int getNumSensors();
- char *getType();
- char *getDriverInfo();

ISISGloveAPI - Glove

La gestione dei valori dei sensori e della calibrazione si effettua semplicemente tramite le funzioni

```
RawHandPosition *getRawHandPosition();

ScaledHandPosition *getScaledHandPosition();

GloveCalibration *getCalibration();

void setCalibration(GloveCalibration*);

void resetCalibration();
```

ISISGloveAPI - HandPosition

- HandPosition
  - Cosi' astratta che per ora non l'ho proprio messa
- RawHandPosition incapsula un insieme di valori raw dei sensori in un certo istante
  - Da usare se per qualche ragione si vuole ignorare/reimplementare lo scaling...
- ScaledHandPosition incapsula un insieme di valori scalati
  - in base all'autocalibrazione dinamica del guanto
    - Che pero' possiamo manipolare come oggetto GloveCalibration
      - GloveCalibration *Glove::getCalibration()
      - void Glove::setCalibration(GloveCalibration* gc)

ISISGloveAPI - Calibration

- GloveCalibration
  - Banalmente
    - incapsula i due array di valori massimi/minimi rappresentanti una calibrazione
    - Fornisce due metodi per il save/load su file

- Volendo....

```
ScaledHandPosition shp;

shp = my_raw_handposition.scaleBy(my_calibration);
```

ISISGloveAPI: TODO?

- Ripulire e rendere l'API meno C e piu' C++
  - Impacchettamento ordinato del tutto in una libreria
  - Uso della stdlib (string e non char*, ecc.)
  - lancio di eccezioni nel caso non si riesca ad aprire il guanto (e altrove?)
  - Overloading dell'operatore == per RawHandPosition e GloveHandPosition
    - Al momento viene fatto un semplicissimo confronto dei valori con una certa tolleranza, ma si può sicuramente fare di meglio (es: calcolare la media degli scarti e regolarsi in base a quella)

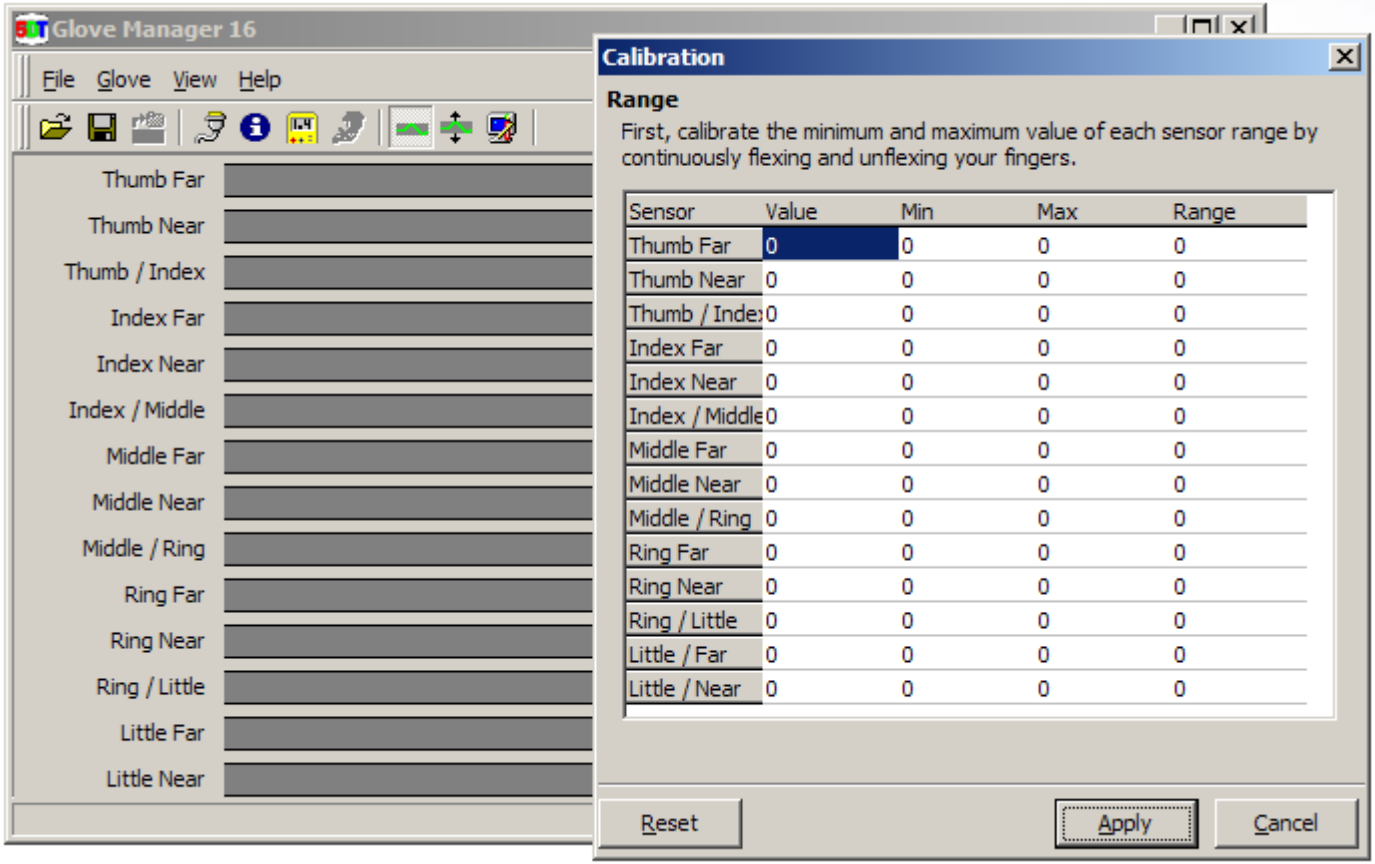
```
bool ScaledHandPosition::compareTo(ScaledHandPosition *other, float toll) {
	for ( int i = 0; i < 18; i++)
		if ( fabs(scaledSensorValues[i] - other->scaledSensorValues[i]) > toll) return
false;
	return true;
}
```

ISISGloveAPI: TODO ??

- Al momento si ha a che fare solo col riconoscimento di posizioni della mano, ma non di movimenti...
- Si potrebbe pensare a una classe HandMovement che contiene una sequenza di HandPosition e la frequenza di campionamento usata per salvarla
  - E a un relativo metodo di riconoscimento, sicuramente più complesso da progettare...
    - tolleranza anche rispetto alla velocità di movimento
    - movimenti che iniziano in maniera simile e poi si diversificano
      - un grafo di HandPosition in cui spostarsi, rilevando i matchs?
- Facile che siano già stati definiti modi diversi (più eleganti) di approcciare e risolvere il problema...

GloveManager16

- Il software fornito é praticamente utile soltanto a provare il guanto...
  - **GloveManager16** (disponibile solo per Windows :( )permette di osservare i valori che arrivano dal guanto, raw o scaled, e di gestirne la calibrazione (con tanto di save/load, come ragionevole...)
  - E poi?



ISISGloveManager?

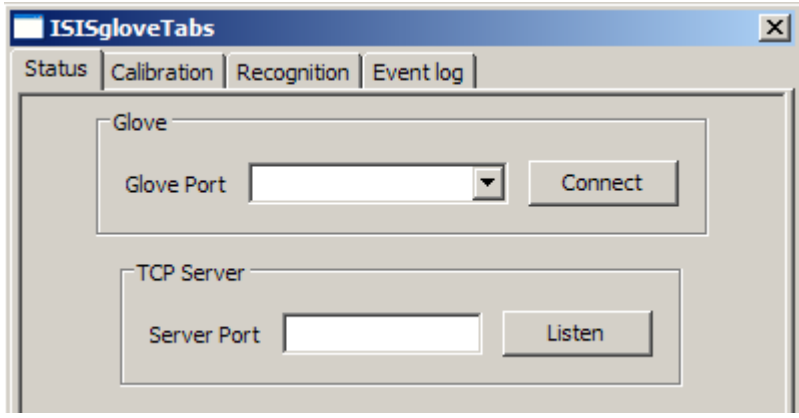
- Sarebbe però utile, gestite le calibrazioni ed eventuali opzioni di scaling, poter accedere ai valori dei sensori o a informazioni semanticamente più significative
  - Esempio: Invio di una segnale “OK” quando viene riconosciuto il pugno chiuso col pollice all'insu'...
- Idea: avere un'utility (cross-platform) di gestione del guanto sempre attiva nella systray, che faccia da tramite tra le applicazioni (magari più di una alla volta) che vogliono accedere al guanto e la nostra GloveAPI in C++...

ISISGloveManager - implementazione

- Come meccanismo di IPC sarei orientato sui socket
  - API e tecniche di programmazione già studiate
  - Tra gli “sviluppi futuri” ci potrebbe essere l'implementazione di qualche dispositivo remoto (wireless?) pilotato dal guanto (grazie a un client per ISISGloveManager a bordo...)
- Altrimenti... memoria condivisa?
  - Non ci ho mai avuto a che fare sotto Win32, ma temo che sia un approccio più OS-dependant...

ISISGloveManager - implementazione

- Per quanto riguarda la GUI dell'utility
  - Insieme al C++ sto iniziando a provare il rodato (in sviluppo attivo dal 1992) toolkit open source wxWidgets
    - sviluppo cross-platform di GUI native, nata per il C++ ma con binding per numerosissimi linguaggi (Perl, Python, Java, Ruby, Lua...)
    - Mi é sembrata dopo varie ricerche una buona scelta, auto-imponendomi sempre il vincolo della portabilità.
  - In effetti sarebbe possibile sviluppare l'applicazione in Java (usando JNI per interfacciarsi con la GloveAPI)
    - Ma chi vuole la JVM in memoria per una piccola utility nella systray?



ISISGloveManager - implementazione

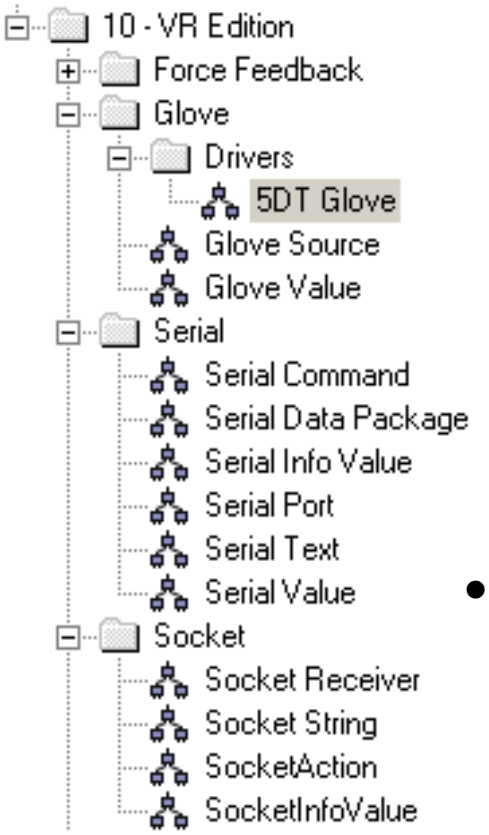
- A questo punto, c'è stata probabilmente qualche battuta su Java...

And now... Quest3D

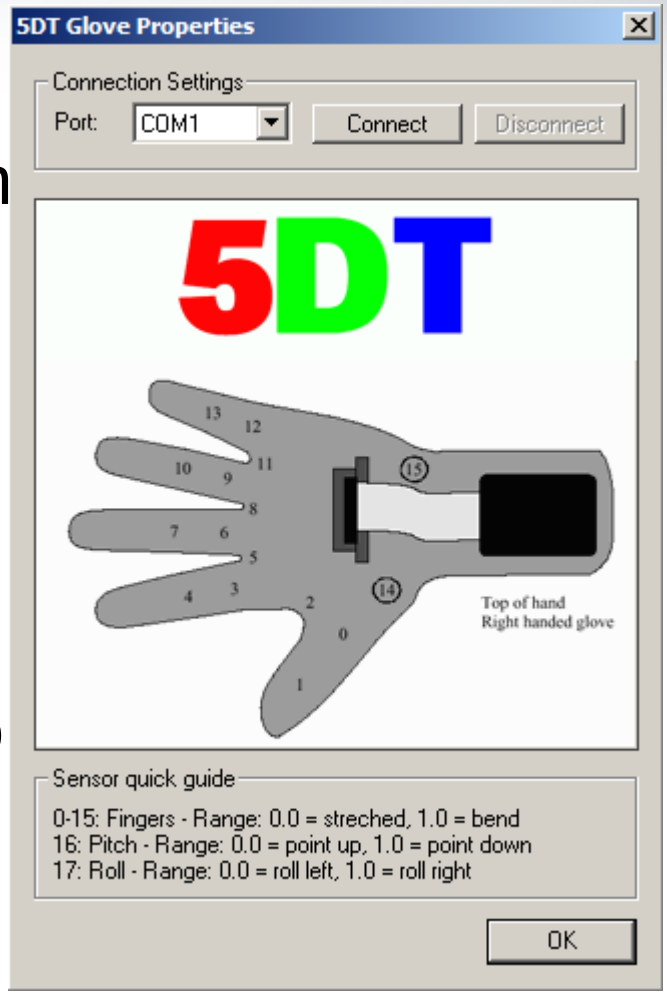
- Spero che non vi siate persi i seminari di Andrea e Luigi!
  - Tralasciamo infatti le caratteristiche generiche di Quest3D per focalizzarci sul supporto dell'engine per i data gloves
  - Vedremo i due approcci con cui ho tentato di utilizzare il guanto in Quest3D
    - Riproduzione del movimento della mano
      - Partendo da un demo incluso in Quest3D
    - Simulazione interattiva dell'uso della mano
      - Partendo da zero e tentando di sfruttare al meglio il motore fisico open source adottato da Quest3D, **ODE**

Quest3D: channels, channels and more channels

- Coerentemente col modello di editing offerto da Quest3D, l'accesso al guanto é fornito tramite degli appositi *channels*...

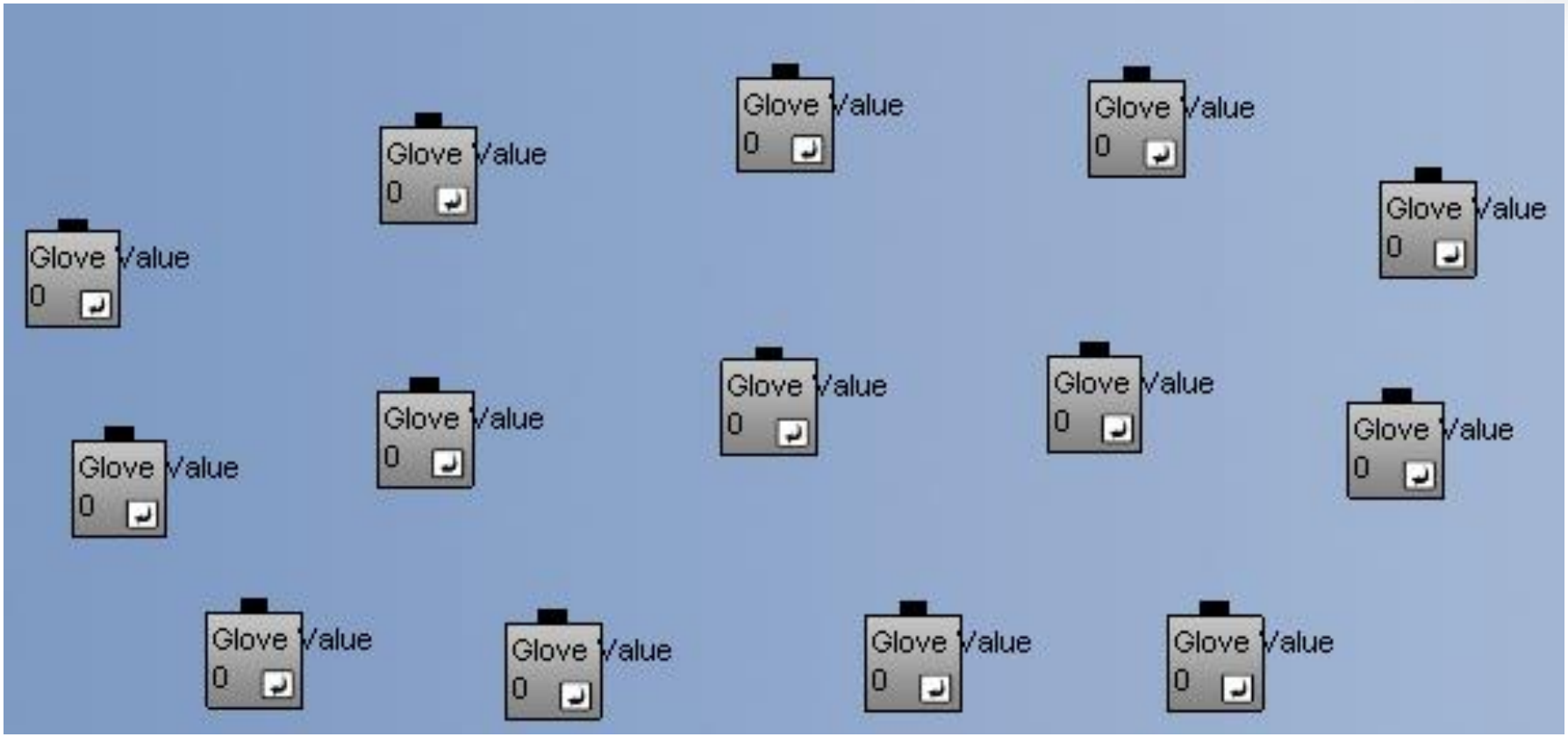


- Quest3D accede alla porta seriale e utilizza il guanto con un suo driver, fornendo dei valori float che presumibilmente dovrebbero corrispondere a quelli scalati in base all'autocalibrazione dinamica del guanto
- Vi ho annoiati al riguardo fino a pochi minuti fa :)



Quest3D: channels, channels and more channels

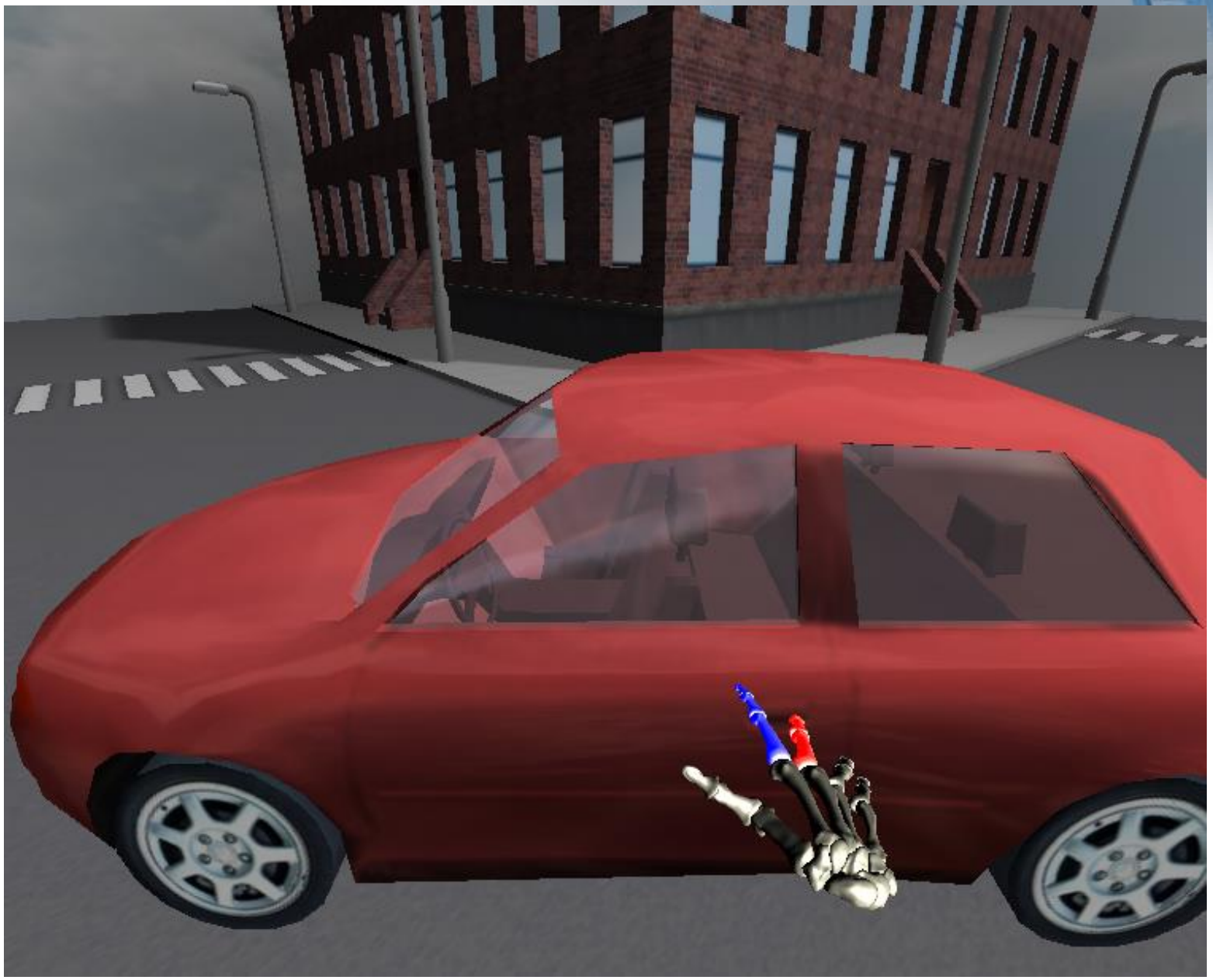
- I channels “Glove Value” permettono di recuperare il valore di un sensore.



Quest3D: Skeleton Hand demo

- Un demo incluso in Quest3D VR edition consiste di una mano scheletrica controllabile dal guanto.
- Il modello della mano usato riproduce fedelmente lo scheletro della mano, ma il demo usa un singolo valore di flessione per i 3 giunti delle dita (2 per il pollice), e non considera la divaricazione
  - In pratica, usa 5 sensori e non 14
- Mi sono limitato a studiarlo e ad ampliarlo per sfruttare tutti i sensori, e ad abbinarlo a una walkthrough camera
  - *Anche se incompleto/migliorabile,vediamolo!*

Quest3D: Skeleton Hand Camera



Quest3D: Skeleton Hand demo

- In sintesi:
  - I valori scalati ottenuti nei channels di Quest3D vengono mappati a un certo spostamento nello spazio 3D delle componenti del modello...
- *“Ok, si muove decentemente... ora attivo la gestione della fisica e ottengo una mano con cui interagire con gli ambienti virtuali....”*
  - Purtroppo non é andata cosi'!
    - Quest3D permette di
      - Usare una semplice gestione delle collisioni basata su sferoidi
      - Usare il motore fisico ODE

Quest3D: la fisica

- Per la simulazione fisica, Quest3D si basa su ODE (Open Dynamics Engine)
- *“ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.”*

  (http://www.ode.org/)

Quest3D: interattività

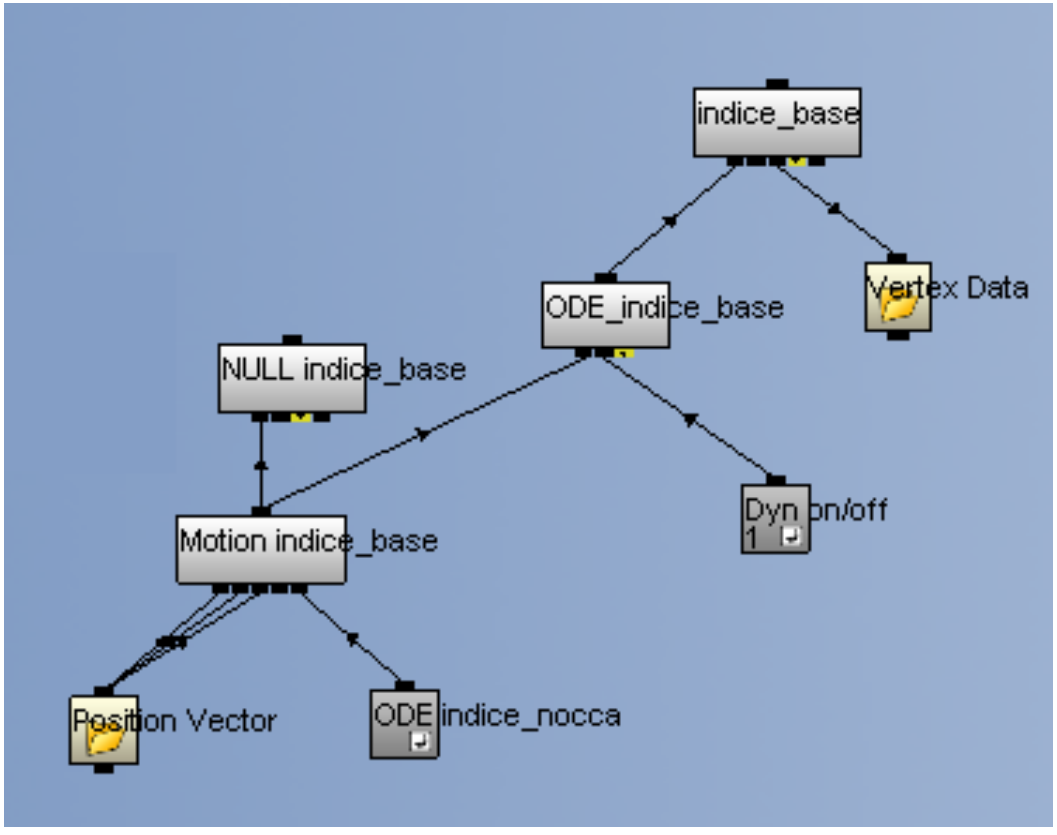
- La “strada” ODE mi è sembrata immediatamente la più interessante...
  - niente animazioni e spostamenti precalcolati/artificiali
  - generica interazione con tutti i corpi ODE (riusabilità)
- Ma...

Quest3D: interattività

- PROBLEMA!
  - Volendo usare il motore fisico, si passa dal dover gestire direttamente le posizioni degli oggetti 3d (valori del loro Motion vector) al doverli spostare applicando delle forze!
    - la posizione degli oggetti viene poi calcolata dal physics engine, considerando tutte le forze in gioco nell'ambiente virtuale
      - Quindi il lavoro sul demo della mano scheletrica non può essere sfruttato per “muovere” una mano “fisicamente attiva”...
      - Cinematica inversa che?

Quest3D: ODE Body

- Un ODE Body rappresenta un corpo rigido per il sistema di simulazione della dinamica
- Viene linkato a un 3D Object, e quando si attiva il motore fisico l'ODE Body controlla il Motion Vector del 3D Object

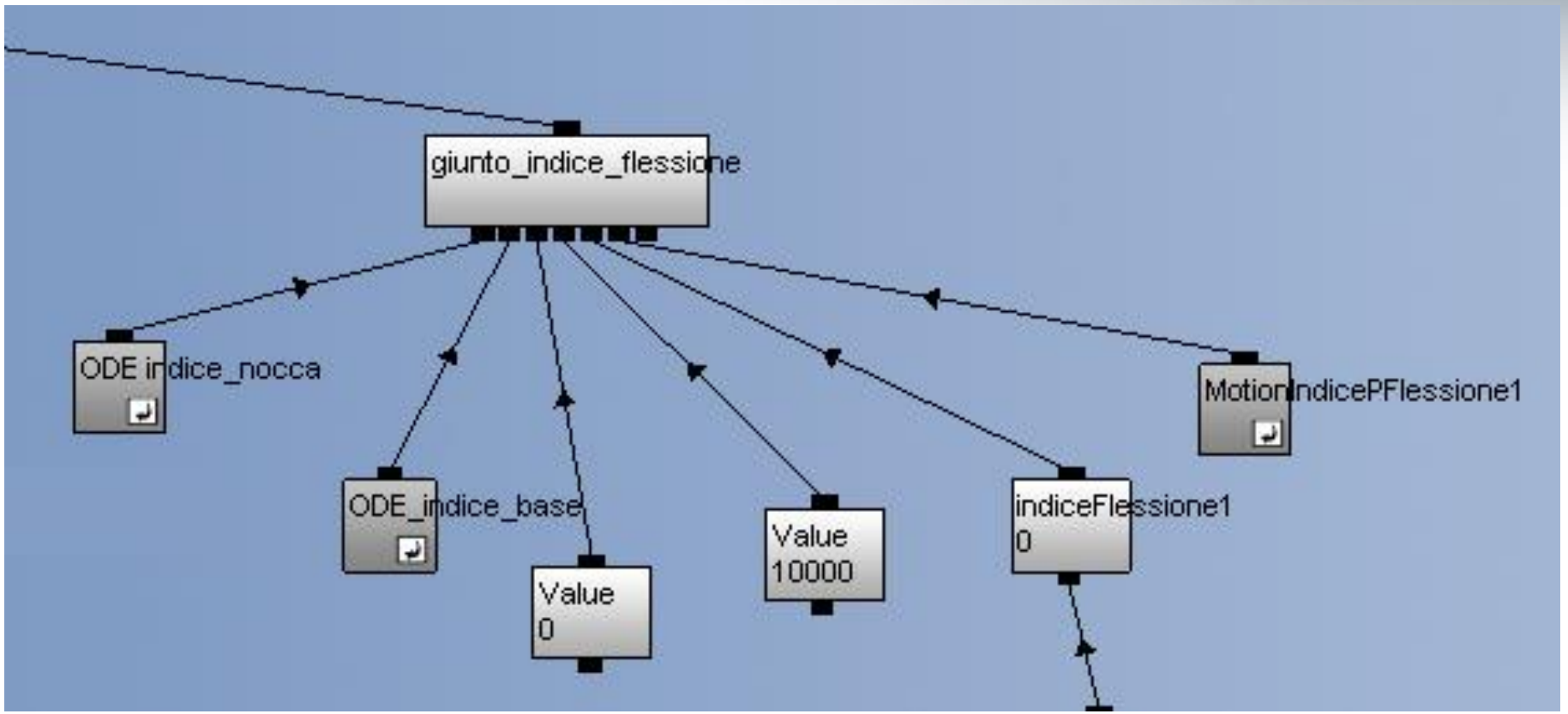


Quest3D: ODE Body

- Ma attenzione....
  - Le geometrie complesse possono essere trattate come ODE Bodies solo per quanto riguarda oggetti statici (es: oggetti ambientali fissi)
  - Per il resto bisogna approssimare le forme con primitive semplici (box, sphere)

Quest3D: ODE Joint

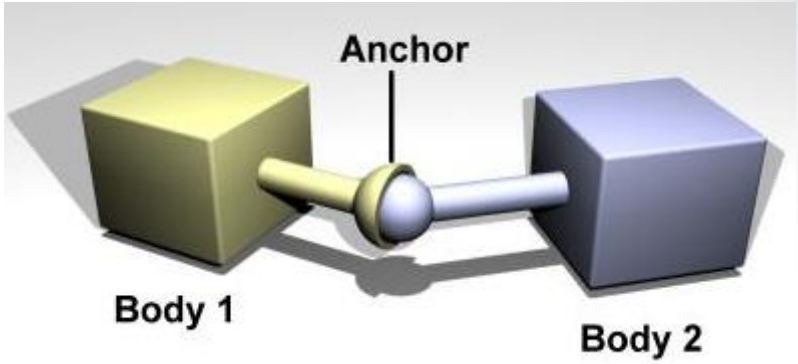
- Un ODE Joint permette di collegare due ODE Body

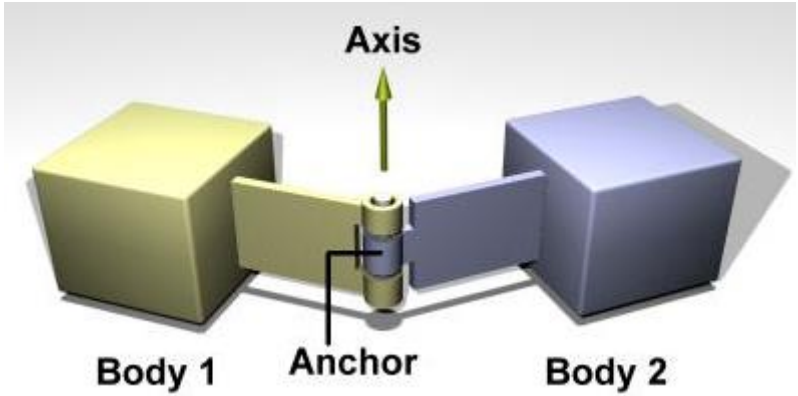


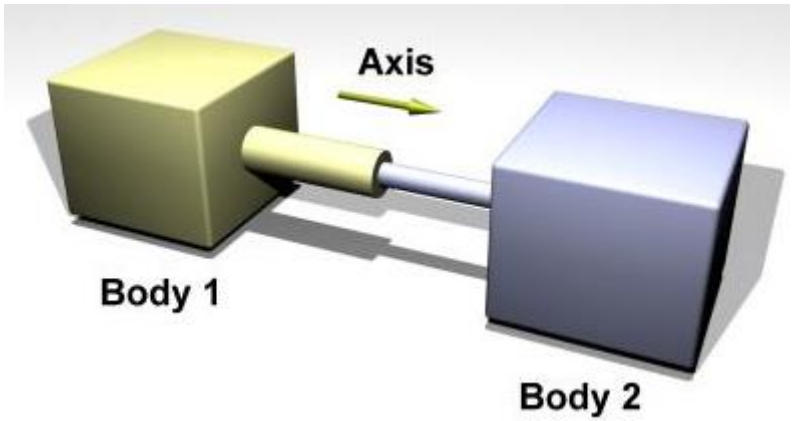
Quest3D: ODE Joint

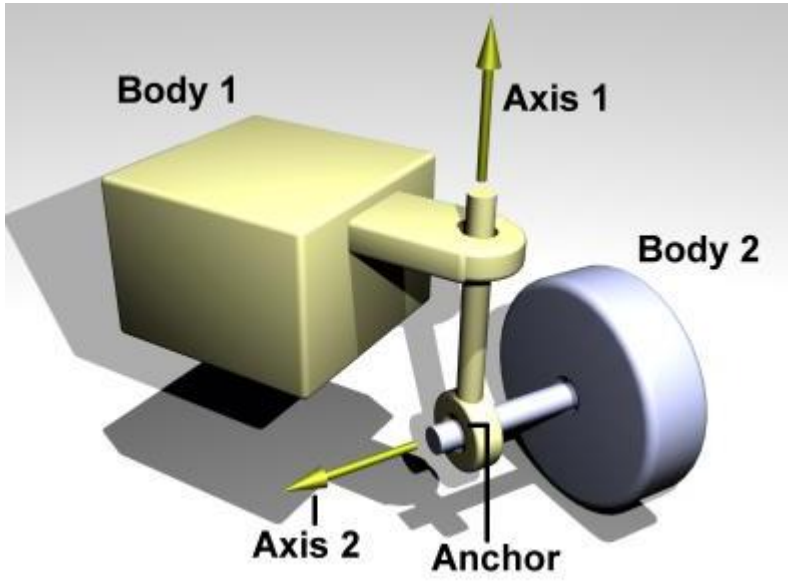
- Regole da osservare lavorando con gli ODE Joints:
  - Due corpi connessi da un giunto non possono collidere tra di loro
  - Due corpi non possono essere connessi l'un l'altro da piu' di un giunto. Piu' giunti tra le stesse due forme si contraddirebbero, comportando una simulazione instabile
  - Determinati tipi di giunto possono vincolare la libertà di movimento in determinati modi (come è naturale aspettarsi).

Quest3D: tipi di ODE Joint







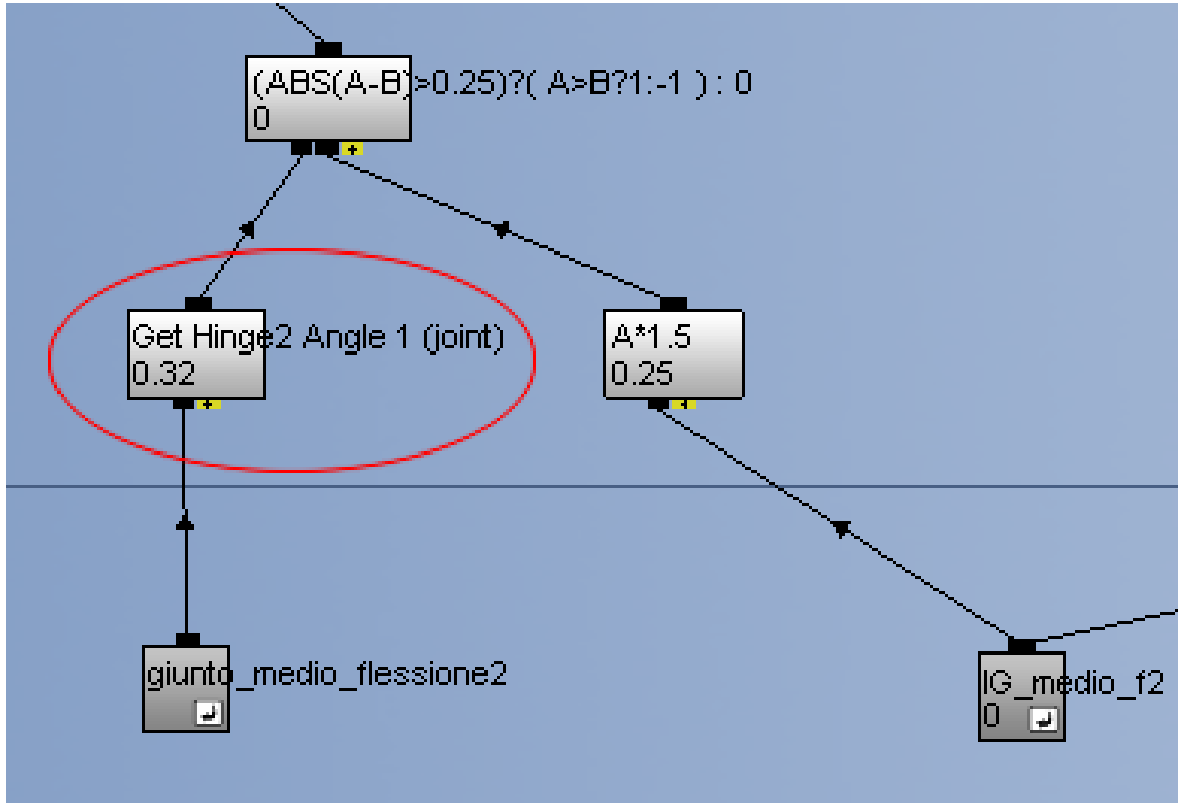


Quest3D: ODE Command

- Il channel “ODE Command” permette di
  - impostare vari parametri globali della simulazione
    - es: *Set Gravity Vector*
  - di agire su uno specifico ODE Body, applicando delle forze
    - *Add Force, Add Torque...*

Quest3D: ODE InfoValue

- Il channel “ODE Info Value” permette di ottenere dal motore fisico informazioni riguardo un ODE Body o un ODE Joint (forza e torsione correntemente applicate, velocità lineare del corpo sui vari assi, informazioni sullo stato dei giunti...)
- *GetLinearVelocity*
- *GetAngularVelocity*
- *GetForce*
- *GetTorque*
- *Get relative point velocity*
- *...*

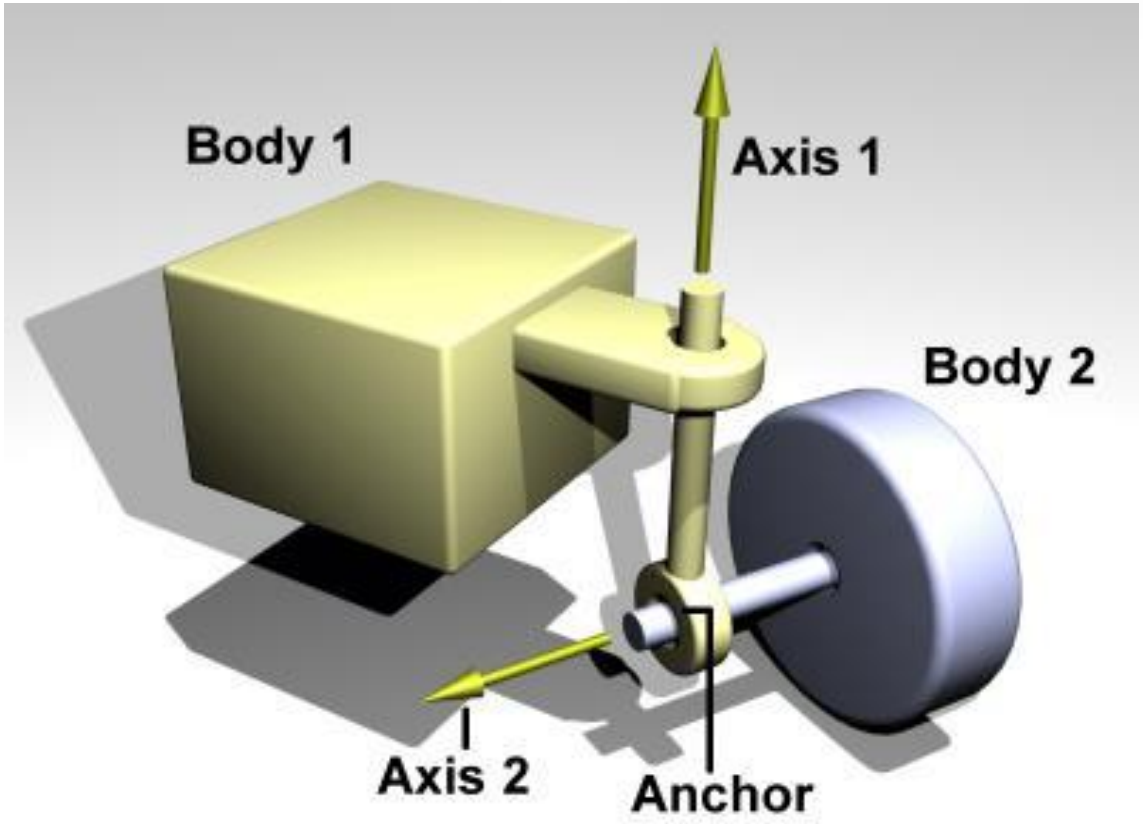


Quest3D: obiettivo ODE hand

- Nuovo target: creare da zero un modello di mano ODE-powered
  - Punto chiave di partenza:
    - Il valore di flessione dato da un sensore deve determinare l'angolo tra due ODE Bodies, che vanno uniti tramite un ODE Joint

Quest3D: Hinge2 Joint

- Appare un po' strambo, ma l'unico giunto che sono riuscito a usare a tale scopo e' quello tipicamente usato per modellare le ruote sterzanti (con sospensioni!) di un veicolo

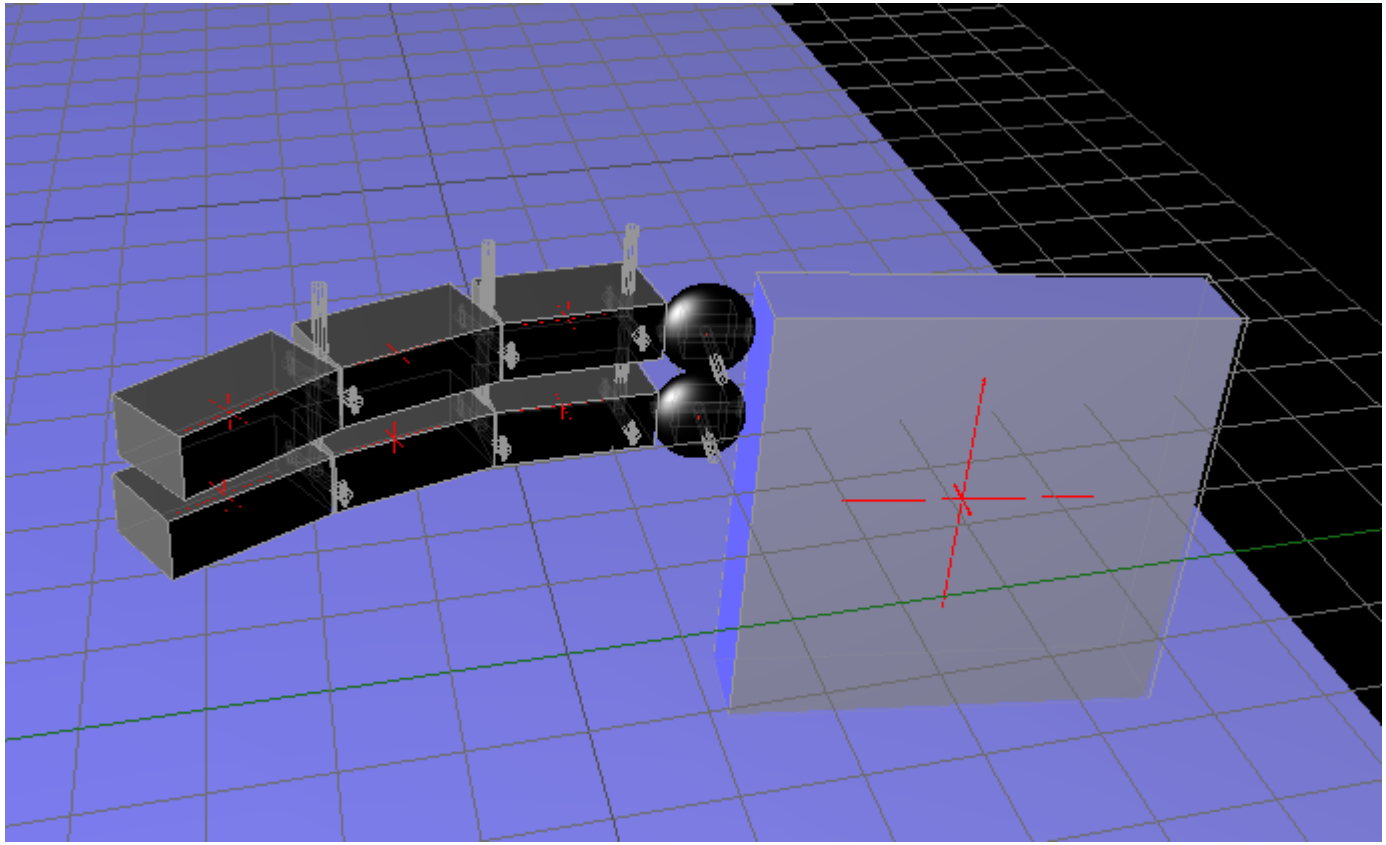


Quest3D: Hinge2 Joint

- Perche'?
  - e' l'unico a fornire l'informazione sull'angolo corrente tra i due corpi rispetto ad un asse, ovvero l'angolo di sterzata (!)
  - In loop, applico una forza per “sterzare” in un senso o nell'altro finché l'angolo tra i due corpi non é adeguato al valore fornito dal sensore, a meno di una tolleranza
    - e' brutto, lo so
      - Si accettano idee :)

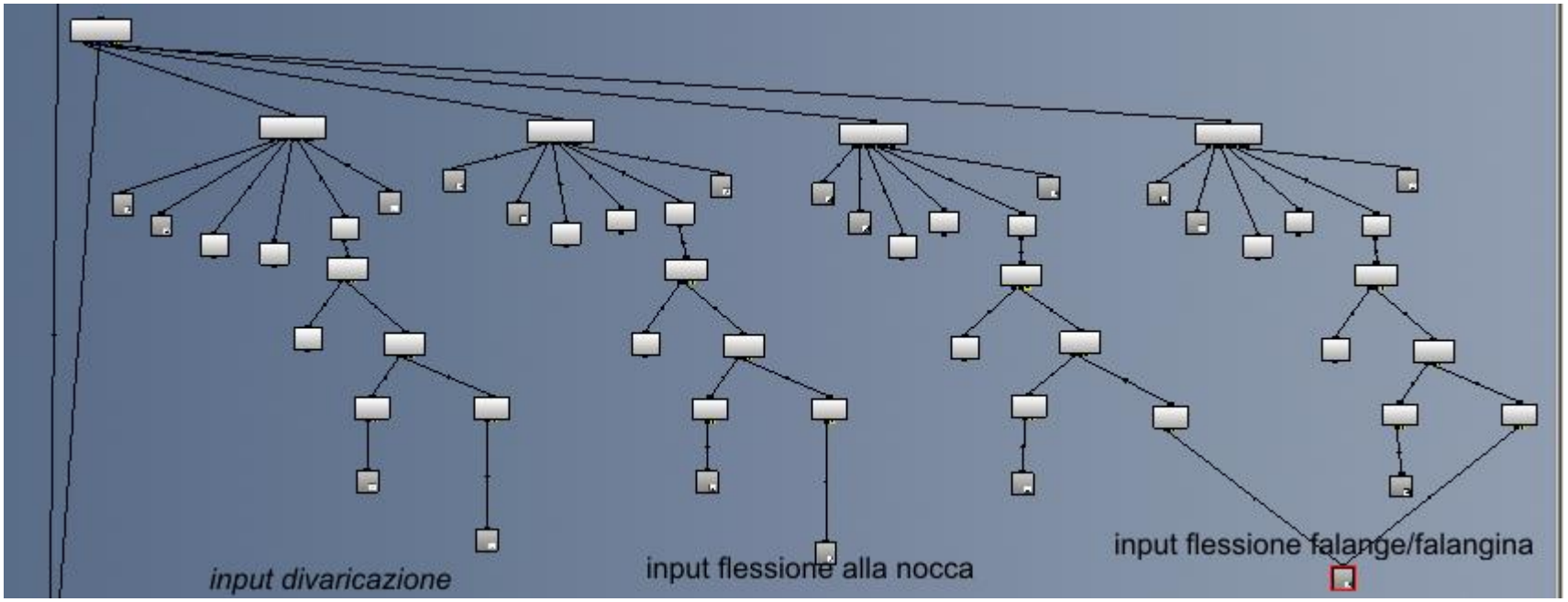
Quest3D: Prima un dito...

- Ho iniziato col modellare un dito con 3 box (falange/ falangina/falangetta) collegati a un box rappresentante il corpo della mano.



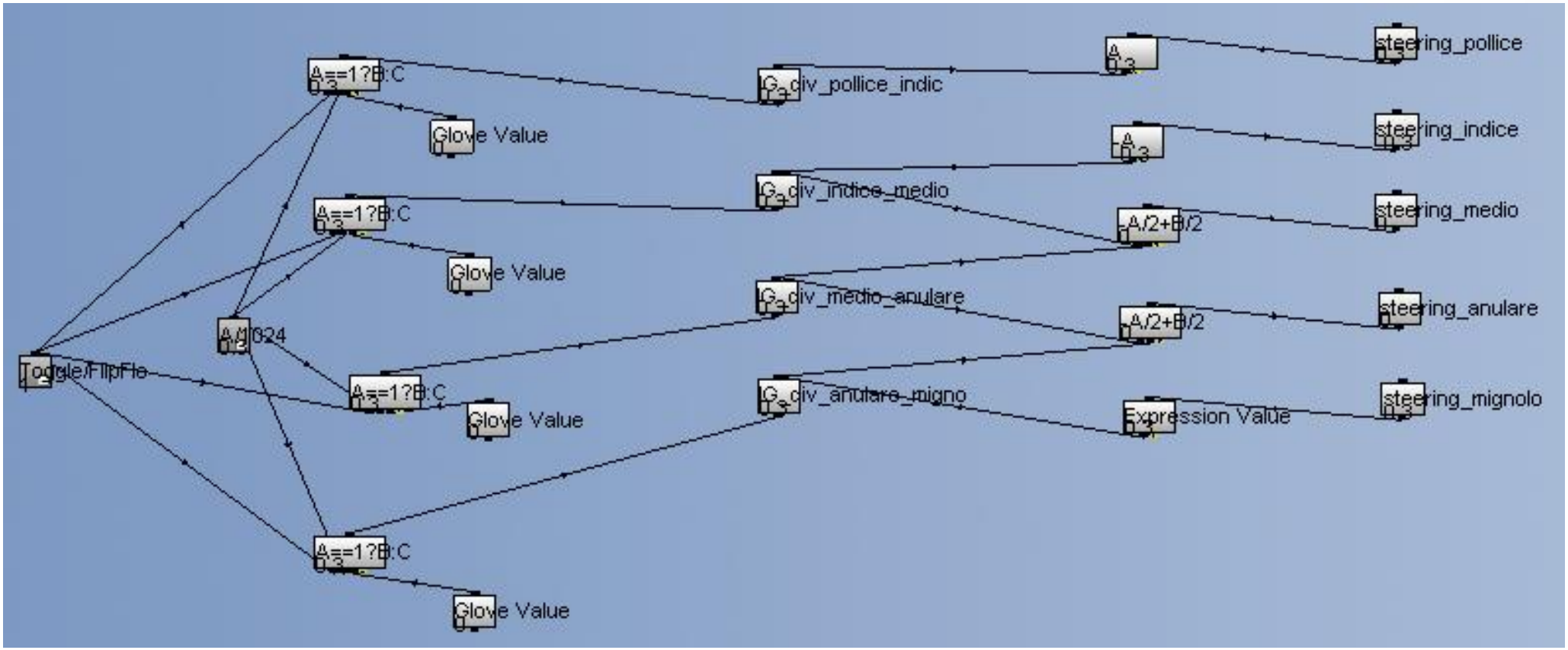
Quest3D: 2 sensori per 3 giunti

- Ogni dito (eccetto il pollice) ha tre punti di flessione, mentre il guanto ne rileva due (alla nocca e tra falange e falangina, ma non tra falangina e falangetta)
  - E' ragionevole che sia cosi' (chi riesce a flettere tra falangina e falangetta senza flettere anche tra falange e falangina?)
    - Quindi il valore di flessione falangina/falangetta fa da input a due giunti



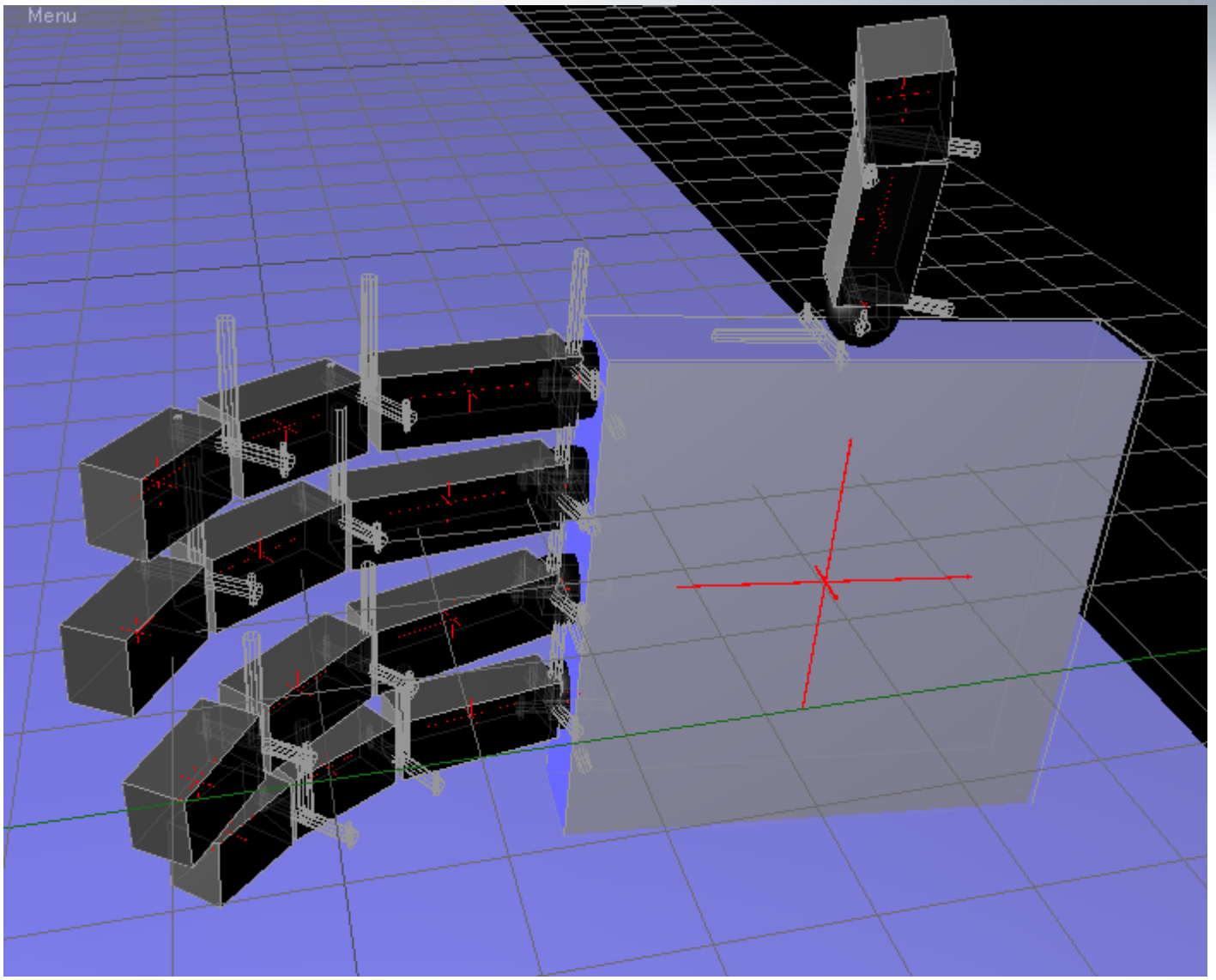
Quest3D: ...e 4 x 5...

- C'e' un sensore di divaricazione tra ogni coppia di dita ma ricordandosi delle regole sull'uso degli ODE Joints si realizza che bisogna usare un giunto per dito.
  - Quindi abbiamo 4 valori a fare da input per 5 giunti



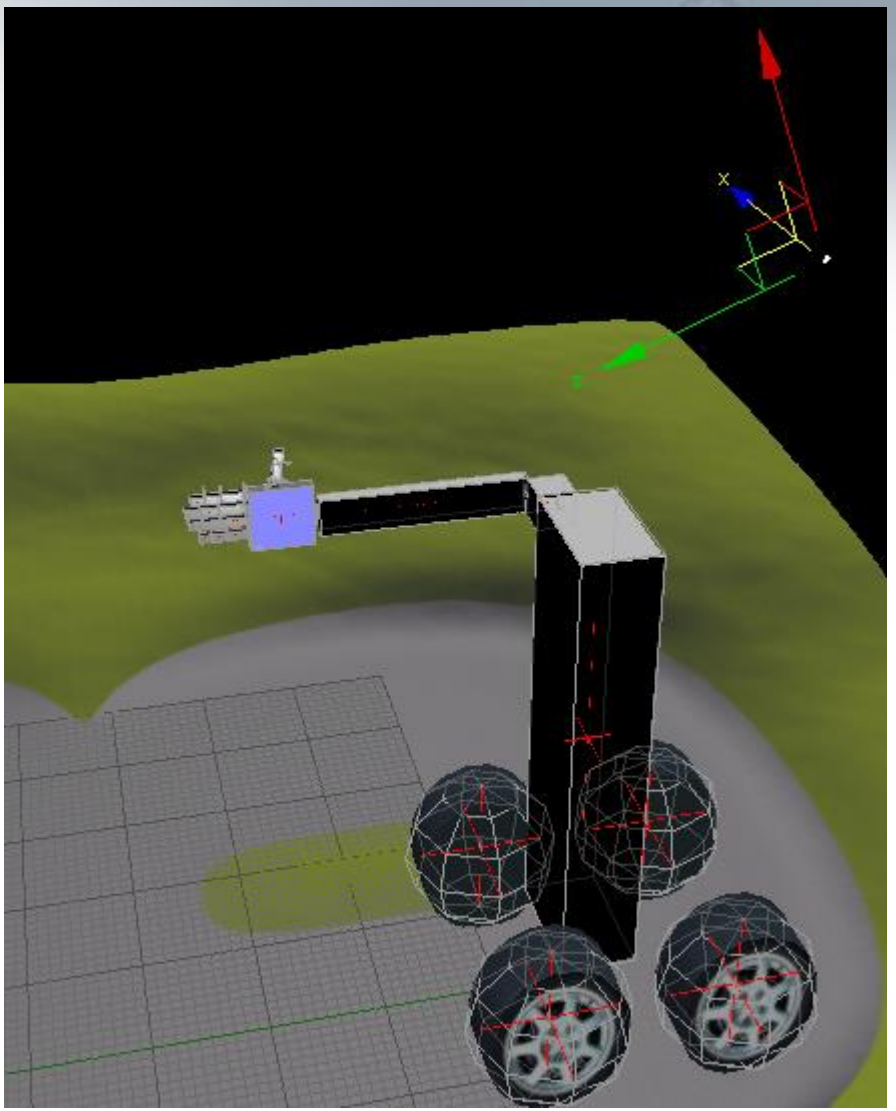
Quest3D: ODE Hand (finalmente)

- Passo passo ho costruito l'intera mano (15 box, 5 sfere, 19 giunti)

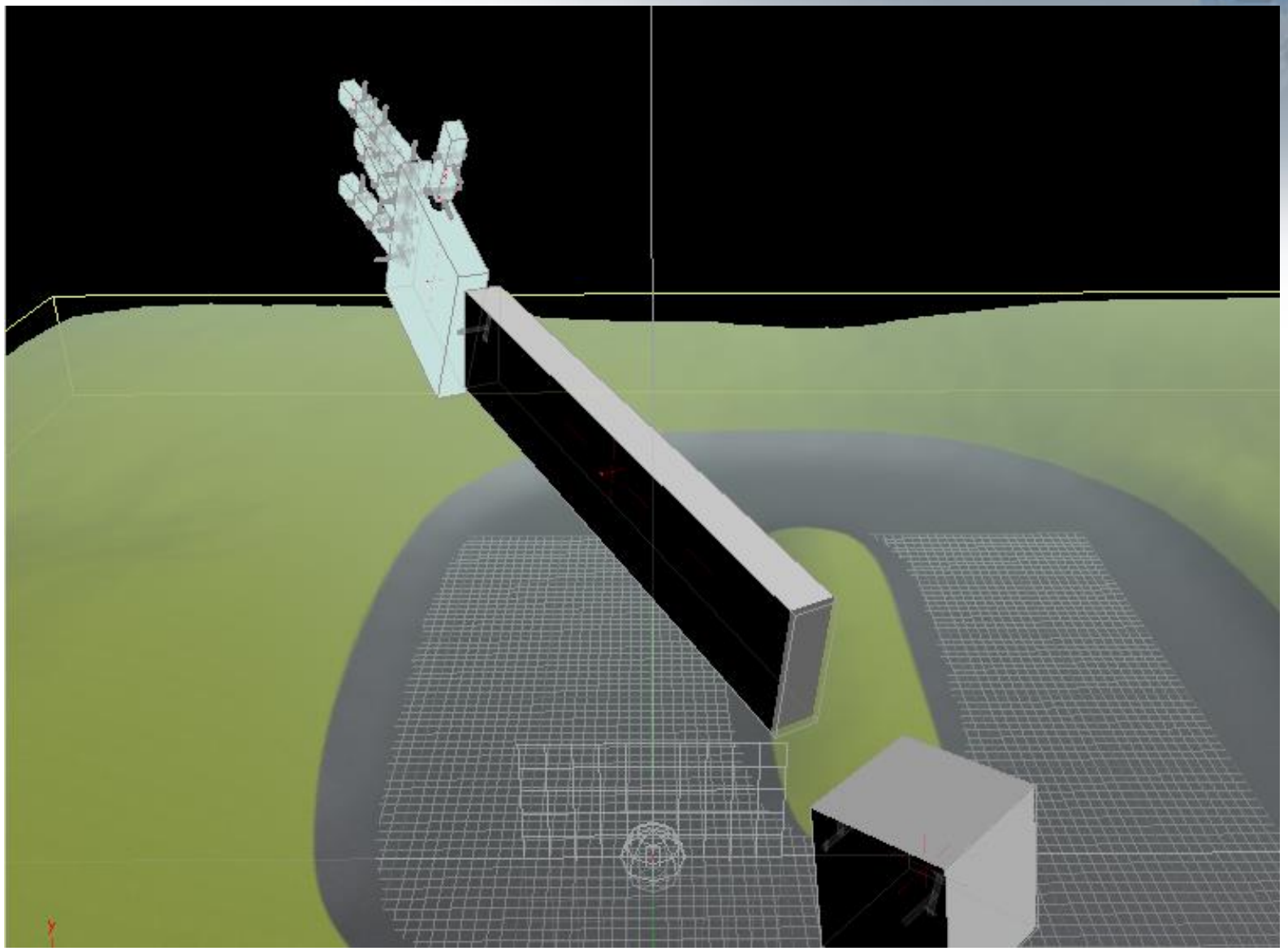


Quest3D: ...e poi?

- Infine ho costruito un supporto mobile su ruote con un “avanbraccio” a cui ho collegato la mano
  - Con una camera che lo segue, posta subito sul retro
- Un po' un accrocco...
  - Ma con qualche modifica (forse) potrebbe andare...



Quest3D: ODE Hand Camera



Quest3D & ISISGloveManager?

- Ok se si tratta di usare i valori dei sensori “direttamente”, per controllare forze/posizioni...
  - Ma implementare la logica di riconoscimento dei gesti o peggio dei movimenti della mano direttamente in Quest sarebbe probabilmente complesso/noioso
    - incubi a base di ragnatele di quadratini

Quest3D & ISISGloveManager?

- Oltre ai generici vantaggi pensati prima per ISISGloveManager, si avrebbe una logica di riconoscimento disaccoppiata
  - Migliorare/cambiare il matching dei gesti senza dover modificare i progetti che li “utilizzano” in Quest3D
  - Mantenere comunque in parallello la possibilità di utilizzare direttamente i valori dei sensori
    - Immaginiamo due connessioni fatte da Quest: una richiede il flusso di valori dei sensori e li usa per animare un modello della mano, un'altra preleva invece dei valori come “ok”,”prendi”,”indica” a cui far corrispondere delle azioni....

Quest3D & ISISGloveManager - come?

- Se si sviluppa ISISGloveManager usando i socket per l'IPC, potrebbe essere sufficiente usare l'accesso alla rete di Quest3D
  - Sono infatti disponibili dei channels per l'utilizzo dei socket
- In ogni caso, si potrebbe sviluppare un Channel “ISISGloveManagerClient” con l'SDK di Quest3D
  - Interessante, ma non é detto che ne valga la pena in questo specifico caso

...FINE! (per ora)

(grazie e alla prossima!)