0.1 programma per l'acquisizione dei dati dal kinect e da arduino

Questo programma è stato usato per acquisire i dati da arduino e dal kinect e scriverli in dei file di testo, ce vengono successivamente elaborati.

0.1.1 librerie usate

La lista completa delle librerie usate è:

```
// libreria usata da visual studio, da togliere in caso si usi un altro ide
#include "pch.h"

// librerie standard per i file stringhe e altro
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <ostream>
#include <string>
#include <math.h>

// librerie create da noi
#include "real_time.h"
#include "kin_file_manager.h"
#include "ard_file_manager.h"
#include "data_structure.h"

// libreria per usare i thread
#include <boost/thread.hpp>

// altre librerie per log, alcuni tipi di dato, un buffer FIFO
#include <boost/log/core.hpp>
#include <boost/call_traits.hpp>
#include <boost/circular_buffer.hpp>
#include <boost/container/vector.hpp>

// altre librerie per il logger
#include <boost/log/attributes.hpp>
#include <boost/log/attributes/scoped_attribute.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/sinks/sync_frontend.hpp>
#include <boost/log/sinks/text_ostream_backend.hpp>
#include <boost/log/sources/basic_logger.hpp>
#include <boost/log/sources/record_ostream.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/utility/setup/common_attributes.hpp>
#include <boost/log/utility/setup/console.hpp>
#include <boost/log/utility/setup/file.hpp>

#include <boost/smart_ptr/make_shared_object.hpp>
#include <boost/smart_ptr/shared_ptr.hpp>

// altre librerie per i thread
#include <boost/thread/condition_variable.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/thread.hpp>

// librereria per il tempo
#include <boost/date_time/posix_time/posix_time.hpp>

// libreria fatta da noi per gestire la seriale
#include "Serial_handler.h"

// librerie di Windows e kinect
#include <Windows.h>
#include <Kinect.h>
#include <Shlobj.h>

```

```
// namespace standard
using namespace std;
```

Listing 1: librerie usate

0.1.2 Librerie custom

Per alleggerire il file "main" abbiamo spostato delle classi in file esterni, che sono a tutti gli effetti delle librerie a se stanti. Abbiamo costruito librerie per: la gestione del tempo, i file manager, le strutture dati e la gestione della seriale.

0.1.2.1 libreria per il tempo

In "real_time.h" ci sono alcune funzioni per prendere il tempo e misurarlo:

```
#pragma once

#ifndef _REAL_TIME_H_
#define _REAL_TIME_H_

#include <boost/chrono.hpp>
#include <boost/timer/timer.hpp>

class real_time
{
private:
  boost::timer::cpu_timer timer;
  boost::timer::cpu_times t;

public:
  /// <summary>
  /// start the time counting
  /// </summary>
  /// <returns></returns>
  void start();

  /// <summary>
  /// return enlapsed time in millisecond
  /// </summary>
  /// <returns></returns>
  float stop();

  /// <summary>
  /// get the time in millisecond
  /// </summary>
  /// <returns></returns>
  uint64_t get_curr_time();

};

#endif // #ifndef _REAL_TIME_H_
```

Listing 2: librerie usate

0.1.2.2 File manager per il kinect

In "kin_file_manager.h" ci sono le funzioni per gestire il file di testo relativo al kinect:

```
#pragma once

#ifndef _KIN_FILE_MANAGER_H_
#define _KIN_FILE_MANAGER_H_

#include <iostream>
#include <ostream>
#include <fstream>
#include <string>

// in questa libreria sono dichiarate le varie strutture dati
```

```
#include "data_structure.h"

class kin_file_manager
{
private:
  std::fstream f;
  std::ios::_Openmode mode;
  unsigned long int line_id = 0;

public:
  // il costruttore provvede a settare il nome del file e la modalit\`a di apertura
  kin_file_manager(std::string file_name, std::ios::_Openmode mode);

  // questa funzione provvede a scrivere un singolo blocco di dati nel file
  void write_data_line(kinect_data dat);

  // funzione che legge un blocco di dati dal file e lo sposta nella struttura dati
  // se si \`e raggiunta la fine del file la funzione ritorna 0, altrimenti ritorna 1.
  bool read_data_line(kinect_data* dat);

  // distruttore della classe, chiude il file
  ~kin_file_manager()
  {
  f.close();
  }

  // funzione che chiude il file
  void close()
  {
  f.close();
  }

};

#endif // #ifndef _KIN_FILE_MANAGER_H_
```

Listing 3: librerie usate

0.1.2.3 File manager per arduino

La libreria "ard_file_manager.h" che contiene le funzioni per la gestione del file di testo relativo ai dati di arduino:

```
#pragma once

#ifndef _ARD_FILE_MANAGER_
#define _ARD_FILE_MANAGER_

#include <iostream>
#include <ostream>
#include <fstream>
#include <string>
#include "data_structure.h"


class ard_file_manager
{
private:
  std::fstream f;
  std::ios::_Openmode mode;
  unsigned long int line_id = 0;

public:
  ard_file_manager(std::string file_name, std::ios::_Openmode mode);

  void write_data_line(arduino_data dat);

  bool read_data_line(arduino_data* dat);

  ~ard_file_manager()
```

```
28    {
29      f.close();
30    }
31
32    void close()
33    {
34      f.close();
35    }
36  };
37
38  #endif //#ifndef _ARD_FILE_MANAGER_
```

Listing 4: librerie usate

Questa libreria è strutturata in modo totalmente simile a quella per la gestione dei file del kinect trattata nel paragrafo 0.1.2.2 con l'unica differenza che la struttura dati utilizzata è relativa ad arduino e non al kinect.

0.1.2.4 strutture dati

La libreria "data_structure.h" è fondamentale e definisce le strutture dati utilizzate nel resto del programma:

```
1  #pragma once
2
3
4  #ifndef _DATA_STRUCTURE_H_
5  #define _DATA_STRUCTURE_H_
6
7  #include <iostream>
8  #include <string>
9  #include <boost/array.hpp>
10
11 // struttura dati per gestire i dati provenienti dal kinect
12 struct kinect_data
13 {
14   static const int number_of_joints = 3;
15
16   // sottostruttura che contiene i dati di ogni giunto: il suo numero, il giunto attaccato ad
       ↪ esso, la posizione e l'angolo rispetto all'altro giunto
17   struct joit_data
18   {
19     int joint_name = -1;
20     int to_joint = -1;
21     float position[3] = { 0, 0, 0 };
22     float angle[3] = { 0, 0, 0 };
23   };
24
25   //Tabelle ad accesso diretto utilizzate per ottimizzare la procedura di estrazione dati dalla
26   //classe kinect_class e la conversione dai punti nello spazio dello scheletro ad angoli
27   //del braccio.
28   int needed_joint[number_of_joints] = { 8, 9, 10 };
29   int to_joint[24] = { -1, -1, -1, -1, -1, -1, -1, -1, 9, 10, -1, -1, -1, -1, -1, -1, -1, -
       ↪ 1, -1, -1, -1, -1, -1 };
30   int to_ref_joint[24] = { -1, -1, -1, -1, -1, -1, -1, -1, 0, 1, 2, -1, -1, -1, -1, -1, -1, -1,
       ↪ -1, -1, -1, -1, -1, -1 }; // tabella di accesso (chiave numero joint kinect) -> (
       ↪ restituisce l-indice del numero del giunto in needed_joint)
31   float ref_Ang_x[number_of_joints] = { 0, 0, 0 }; //per  8, 9
32   float ref_Ang_y[number_of_joints] = { 0, 0, 0 }; //per 8, 9
33   float ref_Ang_z[number_of_joints] = { 180, 180, 0 }; //per 8, 9
34   float mul_fctor_x[number_of_joints] = { 0, 0, 0 }; //per 8, 9
35   float mul_fctor_y[number_of_joints] = { 1, 1, 0 }; //per 8, 9
36   float mul_fctor_z[number_of_joints] = { 1, 1, 0 }; //per 8, 9
37
38   joit_data needed_joints[number_of_joints];
39
40   // tempo esatto in cui i dati sono stati acquisiti
41   uint64_t frame_time = 0;
42
43   // contatore usato nella post elaborazione dei dati
```

```
  uint64_t contatore = 0;

  // funzione per verificare se il frame acquisito \'e parziale o completo
  bool full_frame();

  // funzione per stampare i dati memorizzati
  void print_data();

  float jointAngleX(float *P1, float *P2);

  float jointAngleY(float *P1, float *P2);

  float jointAngleZ(float *P1, float *P2);

  void updateAngles();

  // funzione per convertire il numero del giunto in una stringa
  std::string joint_Enum_ToStr(int n, std::string language);
};

// struttura dati per gestire i dati provenienti da arduino
struct arduino_data
{
  // variabili che rappresentano i dati: dell'accelerometro, del giroscopio, del magnetometro e
      ↪ la temperatura
  float acc_xyz[3];
  float gy_xyz[3];
  float magn_xyz[3];
  float temp;

  // tempo esatto di acquisizione dei dati
  uint64_t frame_time = 0;

  // ?
  uint64_t contatore = 0;

  // funzione che permette di stampare il set di dati attualmente immagazzinato a schermo
  void print_data();
};

// struttura template che gestisce un set di dati del dataset
template<typename type_in, std::size_t N, typename type_out, std::size_t M>
struct dataset_data
{
  boost::array<type_in, N> in;
  boost::array<type_out, M> out;

  // stamapa i dati attualmente immagazzinati
  void print_data();
};

#endif //#ifndef _DATA_STRUCTURE_H_
```

Listing 5: librerie usate

0.1.2.5 Gestione seriale

Infine, l'ultima libreria fatta da noi ospita la classe che gestisce la seriale per la comunicazione tramite bluetooth con arduino:

```
#pragma once

#ifndef _SERIAL_HADLER_H_
#define _SERIAL_HADLER_H_

#include <iostream>
#include <string>
#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/asio/serial_port.hpp>
```

```
11
12
13 class Serial
14 {
15 private:
16   // dichiaro la porta seriale che verra inizializzata nel costruttore
17   boost::asio::serial_port* port;
18
19   // per ricomporre i float ricevuti uso un union
20   union Scomp_float
21   {
22     float n_float;
23     uint32_t n_int;
24     uint8_t n_bytes[4];
25   };
26
27 public:
28   // nel costruttore si inizializza la seriale e si effettua la connessione al dispositivo
       ↪ bluetooth
29   Serial(std::string com);
30
31   // effettua una sincronizzazione iniziale tra arduino ed il pc
32   void sinc();
33
34   // riceve un float da arduino
35   float receive_float();
36
37   // metodo per inviare un carattere
38   void send_char(char ch);
39
40   // metodo per ricevere un carattere
41   char receive_char();
42
43
44   // riceve i dati di acc, gy, magn da arduino
45   // il metodo ritorna
46   //     -> 0 se la trasmissione \'e andata a buon fine
47   //     -> 1 se \'e fallita
48   int receive_data(float* acc_xyz, float* g_xyz, float* magn, float* temp);
49
50   ~Serial()
51   {
52     port->close();
53   }
54 };
55
56 #endif // #ifndef _SERIAL_HADLER_H_
```

Listing 6: librerie usate

Per trasmettere il blocco di dati da arduino al pc abbiabo sviluppato un "protocollo" di comunicazione, cioè i vari dati sono separati da sei tag di controllo che permettono di verificare se il dato è stato trasmesso correttamente o no.
TODO

```
void send_data(float* acc_xyz, float
    ↪ * g_xyz, float* magn, float
    ↪ temp)
{
  send_char('a');
  for (int i = 0; i < 3; i++)
    send_float(acc_xyz[i]);
  send_char('g');
  for (int i = 0; i < 3; i++)
    send_float(g_xyz[i]);
  send_char('m');
  for (int i = 0; i < 3; i++)
    send_float(magn[i]);
  send_char('t');
  send_float(temp);
  send_char(4);
}
```

(a) codice arduino

```
int Serial::receive_data(float*
    ↪ acc_xyz, float* g_xyz, float*
    ↪ magn, float* temp)
{
  if (receive_char() != 'a')
    return 1;
  for (int i = 0; i < 3; i++)
    acc_xyz[i] = receive_float();
  if (receive_char() != 'g')
    return 1;
  for (int i = 0; i < 3; i++)
    g_xyz[i] = receive_float();
  if (receive_char() != 'm')
    return 1;
  for (int i = 0; i < 3; i++)
    magn[i] = receive_float();
  if (receive_char() != 't')
    return 1;
  temp[0] = receive_float();
  if (receive_char() != 4)
    return 1;
  return 0;
}
```

(b) codice che gira sul pc

Figura 1: codice per l'invio e la ricezione dei dati necessari

Nel codice in figura 1a troviamo la funzione per inviare i dati da arduino, mentre in figura 1b troviamo il codice per ricevere gli stessi dati sul pc. Come si può vedere viene inviato un carattere di controllo prima di ogni dato, questo carattere serve per controllare l'integrità della comunicazione e deve essere ricevuto dal pc. Se ciò non accade significa che nella trasmissione è andato perso 1 byte e quindi arduino ed il pc vanno risincronizzati.
TODO

0.1.3 programma principale dell'acquisizione dati

Oltre a quste classi/strutture integrate in delle librerie separate ci sono altre classi/altro nel main, infatti abbiamo: "namespace logger", "class bounded_buffer", "class kinect_class", "class arduino_class", "class ard_handler", "class kin_handler", "void start_aquire_data" e "void acquire_data".

0.1.3.1 Logger

Per prima cosa è bene spiegare il logger, che viene usato nel resto del programma. Questa libreria serve esclusivamente per il debug e ha attributi specifici per lavoare con più thread contemporaneamente in modo asincrono. Serve fondamentalmente a stampare un testo a schermo, ma una volta settate le impostazioni si può aggiungere a quasto testo il nome del thread il tempo esatto in cui il messaggio viene stampato ed evitare che 2 thread stampino contemporaneamente nella console (se lo facessero il testo si mischierebbe e non si capirebbe nulla). La documentazione approfondita su come funziona questa libreria si può trovare qui. Nel nostro caso i settaggi usati sono: un id che identifica il thread, un tag che identifica il nome del trhead, un tag che identifica il livello di severità, il tempo assoluto e la stringa da stampare. Ad esempio se questa riga è posta all'inizio del main e il main thread viene chiamato "main":

```
BOOST_LOG_SEV(slg, logger::normal) << "hello log";
```

Listing 7: librerie usate

Allora viene stampato a schermo un messaggio formattato in questo modo:

```
id->(0x24a8): main < normal >   time->[00:00:00.000404]  -> hello log
```

Listing 8: librerie usate

Ciò consente di debuggare facilmente operazioni asincrone su più trhead. Il codice relativo hai settaggi necessari per ottenere quel tipo di formattazione è il "namespace logger":

```
namespace logger
{
  // definizione dei divesti livelli di severit\`a
  enum severity_level
  {
    normal,
    warning,
    error,
    critical
  };

  // definizione degli attributi usati
  BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", severity_level)
  BOOST_LOG_ATTRIBUTE_KEYWORD(tag_attr, "Tag", std::string)
  BOOST_LOG_ATTRIBUTE_KEYWORD(timeline, "Timeline", boost::log::attributes::timer::value_type)
  BOOST_LOG_ATTRIBUTE_KEYWORD(tread_id, "Tread_id", boost::thread::id)
  BOOST_LOG_ATTRIBUTE_KEYWORD(tread_name, "tread_name", std::string)

  // provvede a sostituire il numero del severity_level con la relativa stringa
  std::ostream& operator<< (std::ostream& strm, severity_level level){...}

  // inizializzazione della formattazione del logger
  void init(){...}

  // definizione dei vari parametri
  boost::log::sources::severity_logger< severity_level > f_init(){...}
  void attr_thread(...){...}
  void attr_time(...){...}
  void attr_tag(...){...}
  void attr_thread_name(...){...}
};
```

Listing 9: librerie usate

Inoltre per funzionare correttamente richiede che all'inizio del main siano posti alcuni settaggi:

```
// inizializzazione logger
logger::init();
auto slg = logger::f_init();
attr_thread(&slg);
attr_time(&slg);

```

```
// imposto il nome del main trhead in "main"
logger::attr_thread_name(&slg, "main");
```

Listing 10: librerie usate

0.1.3.2 Buffer FIFO

Un altra classe usata nel resto del programma è "class bounded_buffer", questa classe ospita funzioni specifiche per immagazzinare temporaneamente dei dati gestendone l'ingresso e l'uscita dal buffer. Fondamentalmente costituisce un registro FIFO indipendente dal tipo di dato che si utilizza. Questo tipo di buffer viene comunemente usato in caso si ha la struttura thread che produce dati - thread che consuma dati: il produttore acquisisce i dati e li inserisce nel buffer, il consumatore estrae i dati e li processa. La dimensione del buffer viene specificata quando lo si inizializza e deve essere sufficentemente grande in modo da garantire la non perdita di dati e il non rallentamento del trhead produttore. Nel nostro caso abbiamo adottato questo tipo di buffer per poter garantire l'asincronicità delle acquisizioni tra arduino e il kinect, così facendo si garantisce la massima velocità tra le acquisizioni dei dati. Il codice relativo a questa classe è stato quasi interamente presto dagli esempi della libreria boost, qui ed è:

```
template <class T>
class bounded_buffer
{
public:

  typedef boost::circular_buffer<T> container_type;
  typedef typename container_type::size_type size_type;
  typedef typename container_type::value_type value_type;
  typedef typename boost::call_traits<value_type>::param_type param_type;

  // nel costruttore si imposta la capacit\'a del buffer
  explicit bounded_buffer(size_type capacity) : m_unread(0), m_container(capacity) {}

  // questa funzione permette di aggiungere un elemeto al buffer
  void push_front(param_type item){...}

  // questa funzione permette di estrarre un elemento dal buffer
  void pop_back(value_type* pItem){...}

  // restituisce la percentuale di riempimento del buffer
  float percentage_of_filling(){...}

  // svuota il buffer
  void flush(){...}

private:
  bounded_buffer(const bounded_buffer&);             // Disabled copy constructor
  bounded_buffer& operator = (const bounded_buffer&); // Disabled assign operator

  bool is_not_empty() const { return m_unread > 0; }

  bool is_not_full() const { return m_unread < m_container.capacity(); }

  size_type m_unread;
  container_type m_container;
  boost::mutex m_mutex;
  boost::condition_variable m_not_empty;
  boost::condition_variable m_not_full;
};
```

Listing 11: librerie usate

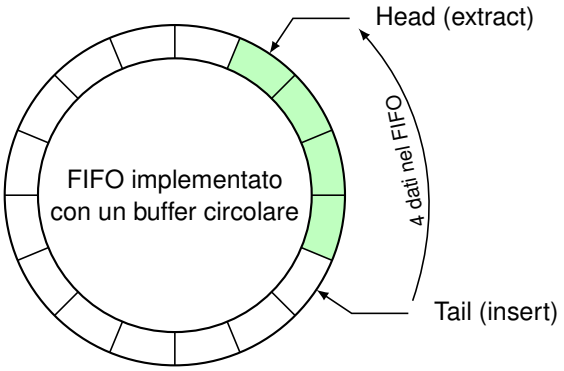

Figura 2: Schema di funzionamento circular buffer

TODO
La classe nel listato 11 utilizza un buffer circolare per implementare il registro FIFO, il funzionamento di questo buffer è descritto dalla figura **??**: ogni volta che si aggiunge un dato viene occupata una locazione in testa al buffer ed ogni volta che si rimuove un dato viene liberata una locazione in coda, cioò permette di riusare continuamente le stesse locazioni di memoria.

0.1.3.3 Classe gestione kinect

La classe "kinect_class" provvede a leggere i dati dal kinect ed aggiungerli al suo corrispettivo registro FIFO. L'estrazione dei dati dal kinect è stata realizzata sulla base dell'esempio body basic D2D integrato nell'sdk browser fornito da microsoft. Il codice relativo a questa classe è:

```
1  class kinect_class
2  {
3  public:
4
5    // Current Kinect
6    IKinectSensor* m_pKinectSensor;
7    ICoordinateMapper* m_pCoordinateMapper;
8
9    // Body reader
10   IBodyFrameReader* m_pBodyFrameReader;
11
12   real_time t;
13
14   bounded_buffer<kinect_data>* FIFO_kin;
15
16   kinect_class(bounded_buffer<kinect_data>* data_FIFO){...}
17
18   void start()
19   {
20
21     bool flag = false;
22     while (true)
23     {
24       {...}
25
26       // prendo il tempo
27       uint64_t t1 = t.get_curr_time();
28
29       // acquisisco i dati e li sposto in data
30       flag = Update(&data);
31
32       // se l'acquisizione \'e andata a buon fine procedo
33       if (flag)
34       {
35         // prendo un altra volta il tempo
36         uint64_t t2 = t.get_curr_time();
37         // cos\'i da poter impostare che il tempo in cui sono stati acquisiti i dati sia
38         // la media tra il tempo prima di cominciare l'acquisizione ed il tempo dopo che essa \'
               ↪ e finita
39         data.frame_time = (t1 + t2) / 2;
40
```

```
41          // inserisco i dati cos\'i acquisiti nel registro fifo
42          FIFO_kin->push_front(data);
43
44          // faccio aspettare un po di tempo a questo trhead perch\'e altrimenti il kinect da
                ↪ errore,
45          // dato che non supporta un acquisizione a pi\'u di 30fps
46          boost::this_thread::sleep(boost::posix_time::milliseconds(35));  al kinect scoppia
47
48        }
49
50        {...}
51
52      }
53    }
54
55    // inizializza il kinect, ritorna 1 se ha avuto successo, 0 altrimenti
56    int InitializeDefaultSensor(){...}
57
58    // acquisisce i dati e li passa alla funzione per processarli
59    bool Update(kinect_data* data)
60    {
61      {...}
62
63      flag = ProcessBody(BODY_COUNT, ppBodies, data);
64
65      {...}
66
67      return flag;
68
69    }
70
71    // questa funzione provvede a spostare i dati da joints a data sistemandoli nella loro
          ↪ struttura dati,
72    // una spiegazione di come ci\'o viene fatto verra data nel capitolo successivo
73    bool handle_data(Joint* joints, kinect_data* data){...}
74
75
76    // quetsta funzione separa le diverse persone rilevate e trasferisce le informazioni
77    // relative hai giunti della prima persona alla funzione handle_data
78    bool ProcessBody(int nBodyCount, IBody** ppBodies, kinect_data* data)
79    {
80      {...}
81
82      for (int i = 0; i < nBodyCount; ++i)
83      {
84        {...}
85
86        Joint joints[JointType_Count];
87
88        hr = pBody->GetJoints(_countof(joints), joints);
89
90        return handle_data(joints, data);
91
92        {...}
93
94      }
95
96      {...}
97    }
98
99    // provvede a spegnere il kinect e deallocare le sue risorse nel caso la classe venga
          ↪ distrutta
100   ~kinect_class(){...}
101   template<class Interface>
102   inline void SafeRelease(Interface *& pInterfaceToRelease){...}
103
104 };
```

Listing 12: librerie usate

0.1.3.4 Classe gestione arduino

La classe "arduino_class" provvede a caricare continuamente i dati provenienti dal kineck nel registro fifo, infatti la funzione "start" viene lanciata come thread e continua a caricare i dati fino al termine del programma:

```
class arduino_class
{
public:
  Serial* ser;
  real_time t;
  bounded_buffer<arduino_data>* FIFO_ard;

  // il costruttore provvede ad inizializzare la seriale e sincronizzare i 2 dispositivi
  arduino_class(bounded_buffer<arduino_data>* data_FIFO, string com_port)
  {
    ser = new Serial(com_port);
    cout << "serial ok" << endl;
    FIFO_ard = data_FIFO;
    ser->sinc();
  }

  // questa funzione viene lanciata all'avvio del thread e provvede ad acquisire i dati ed
      ↪ inserirli nel registro fifo
  void start()
  {
    while (true)
    {
      arduino_data data;

      uint64_t t1 = t.get_curr_time();
      ser->receive_data(data.acc_xyz, data.gy_xyz, data.magn_xyz, &data.temp);
      uint64_t t2 = t.get_curr_time();

    // per ottenere un accurata misurazione del tempo viene preso prima e dopo l'acquisizione e
        ↪ poi fatta la media dei valori ottenuti
      data.frame_time = (t1 + t2) / 2;

      FIFO_ard->push_front(data);
    }
  }
};
```

Listing 13: librerie usate

0.1.3.5 Gestore di arduino e del kinect

Dopo i 2 thread producer si hanno i 2 thread consumer: "class ard_handler" e "class kin_handler", che si occupano rispettivamente di salvare su file di testo le informazioni di arduino e del kinect.

```
/// <summary>
/// handle the data from arduino and write it on a file
/// </summary>
class ard_handler
{
private:
  bounded_buffer<arduino_data>* ard;
  ard_file_manager* f;

public:
  // il costruttore provvede ad inizializzare la classe per gestire i file di testo di arduino
  ard_handler(bounded_buffer<arduino_data>* ard, string file_name)
  {
    this->ard = ard;
    f = new ard_file_manager(file_name, std::ios::out);
  }

  // questa funzione \'e quella che viene effettivamente lanciata all'avvio del thread e
      ↪ provvede a scaricare il buffer scrivendo i dati sul file fino al raggiungimento della
      ↪ quota predichiarata "tot_esempi"
```

```
  void start()
  {
  // inizializzazione logger
    auto slg = logger::f_init();
    attr_thread(&slg);
    logger::attr_thread_name(&slg, "ard_handler");

    BOOST_LOG_SEV(slg, logger::normal) << "hello log";

    for (int i = 0; i < tot_esempi; i++)
    {
      arduino_data data_ard;
      ard->pop_back(&data_ard);

      f->write_data_line(data_ard);

    }

  // segnalo sulla console qunado il thread ha finito il suo lavoro
    BOOST_LOG_SEV(slg, logger::normal) << "end";
  }
};
```

Listing 14: librerie usate

La classe "kin_handler" esegue esattamente le stesse operazioni della precedente con la sola differenza che usa "kin_file_manager" al posto di "ard_file_manager".

```
/// <summary>
/// handle the data from the kinect and write it in the file
/// </summary>
class kin_handler
{
private:
  bounded_buffer<kinect_data>* kin;
  kin_file_manager* f;

public:
  kin_handler(bounded_buffer<kinect_data>* kin, string file_name){...}

  void start(){...}
};
```

Listing 15: librerie usate

0.1.3.6 Funzioni di lancio dei thead per l'acquisizione ed il salvataggio dei dati

Dopo tutte queste classi si hanno 2 funzioni che si occupano di inizializzare le altre classi, lanciare tutti i thread neccessari ed acquisire dall'utente i nomi dei file etc.

```
void start_aquire_data(string out_ard, string out_kin, string com_port)
{
  // in inizializzo il logger per questa funzione
  auto slg = logger::f_init();
  attr_thread(&slg);
  logger::attr_thread_name(&slg, "main");

  // dichiaro i 2 buffer FIFO
  bounded_buffer<arduino_data> data_FIFO_ard(20);
  bounded_buffer<kinect_data> data_FIFO_kin(20);

  // chiamo i costruttori delle classi che gestiscono il kinect e arduino
  // al costruttore di arduino gli devo passare la com a cui il bluetooth \'e collegato
  arduino_class a(&data_FIFO_ard, com_port);
  kinect_class k(&data_FIFO_kin);

  // chiamo i costruttori delle classi per la scrittura nei file di testo
  ard_handler a_h(&data_FIFO_ard, out_ard);
  kin_handler k_h(&data_FIFO_kin, out_kin);
```

```
20
21    BOOST_LOG_SEV(slg, logger::normal) << "costructor end" << std::endl;
22
23    // lancio i thread che "producono i dati"
24    boost::thread ard([&a]() { a.start(); });
25    boost::thread kin([&k]() { k.start(); });
26
27    BOOST_LOG_SEV(slg, logger::normal) << "producer thread launced" << std::endl;
28
29    // lancio i thread che "consumnano dati"
30    boost::thread ard_h([&a_h]() { a_h.start(); });
31    boost::thread kin_h([&k_h]() { k_h.start(); });
32
33    BOOST_LOG_SEV(slg, logger::normal) << "consumer thread launched" << std::endl;
34
35    // aspetto all'infinito (finch\'e il programma non viene chiuso
36    kin.join();
37    ard.join();
38    kin_h.join();
39    ard_h.join();
40 }
```

Listing 16: librerie usate

infine c'è la funzione che si occupa di acquisire i dati dall'utente e passarli alla funzione appena vista.

```
1  void acquire_data()
2  {
3    string ard_file_mane = "";
4    string kin_file_mane = "";
5    string arduino_com_port = "";
6
7    // acquisisco il nome dei 2 file che conterranno i dati di arduino e i dati del kinect
8    cout << "insert the file names: " << endl << "\t arduino -> ";
9    std::getline(std::cin, ard_file_mane);
10   cout << "\t kinekt -> ";
11   std::getline(std::cin, kin_file_mane);
12
13   // acquisisco il numero della com a cui \'e collegato il bluetooth del computer
14   cout << "insert the arduino com port -> : ";
15   std::getline(std::cin, arduino_com_port);
16
17   // lancio la funzione passandogli i dati appena acquisiti
18   start_aquire_data(ard_file_mane, kin_file_mane, arduino_com_port);
19 }
```

Listing 17: librerie usate

0.1.3.7 Main

Infine il main chiama semplicemente la funzione appena trattata.

```
1  int main()
2  {
3    logger::init();
4
5    auto slg = logger::f_init();
6    attr_thread(&slg);
7    attr_time(&slg);
8    logger::attr_thread_name(&slg, "main");
9
10   BOOST_LOG_SEV(slg, logger::normal) << "hello log";
11
12   acquire_data();
13
14   return 0;
15 }
```

Listing 18: librerie usate

0.1.4 Riassunto

Riassumendo la struttura generale di questo programma si ha:

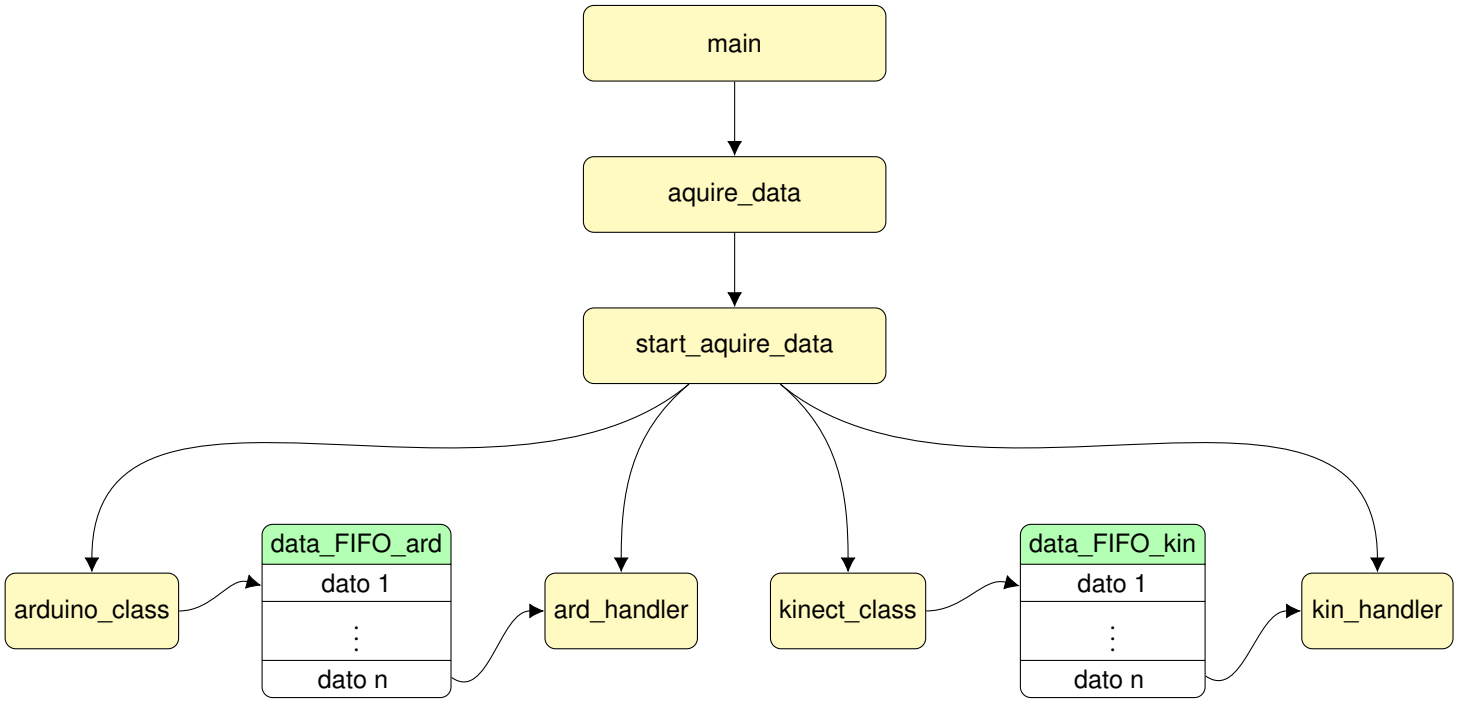

Figura 3: i programmi usati per ottenere i pesi della rete addestrata (TODO?)

Cioè il "main" chiama "aquire_data" che chiama a sua volta "start_aquire_data". Dopodichè vengono lanciati i 4 thread ("arduino_class", "ard_handler", "kinect_class", "kin_handler"). I thread che si occupano di prelevare i dati dai dispositivi sono "arduino_class" e "kinect_class", che inseriscono i dati nel corrispettivo buffer FIFO. Infine i thread che si occupano di scrivere i dati su i file di testo sono "ard_handler" e "kin_handler", che scrivono i dati prelevati dal buffer nel corrispettivo file di testo.