Appendice B

Listati

Vengono di seguito riportati i listati relativi agli aspetti fondamentali del progetto da noi sviluppato, ossia il *sistema di visione* ed il *processo d'interazione*, implementati rispettivamente all'interno del modulo *head-analyzer* ed *e2-brain*.

B.1 Il Sistema di Visione: il modulo *head-analyzer*

```
/*=================================================
 Authors: cristianmandelli@gmail.com
        deborahzamponi@gmail.com
 Data: 15/10/2011
 Description: Detect face features on a video captured
    from Kinect
=================================================*/

#include "ros/ros.h"
#include "ros/package.h"
#include "opencv2/opencv.hpp"
#include "std_msgs/String.h"
#include "geometry_msgs/Point32.h"
#include <string.h>
#include <cv_bridge/cv_bridge.h>
#include <cv_bridge/CvBridge.h>
#include <image_transport/image_transport.h>
#include <math.h>
#include <set>
#include "Blob.h"
#include "Classifiers.h"

//Ros messages
```

```
#include "user_tracker/Com.h"
#include "head_analyzer/MoveDataMSG.h"
#include "head_analyzer/HeadDataMSG.h"
#include "head_analyzer/EyesDataMSG.h"

#define USER_DISTANCE 1000
#define EYES_TRACKING_DISTANCE 850
#define FRAME_WIDTH 640
#define FRAME_HEIGHT 480

#define BLOB_HEAD_WIDTH 200
#define BLOB_HEAD_HEIGHT 200

#define BLOB_EYE_RATIO 0.9
#define MAX_EYE_CANDIDATE_SIZE 1

#define MIN_MOVES 3      //min num of samples in an
   action
#define MAX_MOVES 20     //max num of samples in an
   action
#define MAX_NO_MOVES 2     //max num of holes in an
   action
#define DENYING_INTESITY 15    //max intensity value of
   denying and nodding

#define DELTA 15
#define SIGMA 20

#define INTERPOLATION_FACTOR 20

unsigned int MIN_EYE_BLOB_SIZE;
unsigned int MAX_EYE_BLOB_SIZE;

const unsigned int FEATURES_NUMBER = 20; //max num of
   features to find in optical flow analysis
const unsigned int THRESHOLD_HEAD_NEAR = 220;  //190
const unsigned int THRESHOLD_HEAD_FAR = 150; //12s0
          //High Light -> 200;
          //Low Light  -> 150;

static const double pi = 3.14159265358979323846;

using namespace cv;
using namespace std;
```

```
//Structures
//=================================================
//=================================================
struct pointer
{
 CvPoint pt1; //starting point of arrow
 CvPoint pt2; //ending point of arrow
};

//Global Variables
//=================================================
//=================================================
bool getFrame;    //Get new frame only if current frame
   analysis is done
bool IRImageReady;
bool depthImageReady;
bool userDistanceReady;
bool detectMovesEnable;
bool eyesFilteringEnable;
bool headDataMessageReady;
bool eyesDataMessageReady;
bool moveDataMessageReady;

int userDistance;
int userHeadPitch;
int userHeadRoll;
int countHolesMove;

float userEyeLratio;
float userEyeRratio;
float userHeadRatio;

double moveIntensity;

unsigned char userMoveClass;

string movesClassifier;

CvPoint headPosition, headPositionPrev;

Mat frameIR, prevFrameIR;
Mat frameDepth;
Mat frameDrawn;

Blob headBlob;
```

```
Blob faceBlob;
Blob eyesAreaBlob;

Rect leftEye, rightEye;

Array1D2DPointDataset move;  //array for saving each move
    that create a movement
Array1D2DPointDataset movement; //array for saving data
   of a complete movement
KNNClassifier *knn;
vector<int> knnScores;
vector<char> knnClass;

sensor_msgs::CvBridge img_bridge_;
ros::Publisher pubHeadData;
ros::Publisher pubEyesData;
ros::Publisher pubMoveData;
head_analyzer::HeadDataMSG headData;
head_analyzer::EyesDataMSG eyesData;
head_analyzer::MoveDataMSG moveData;


//Prototypes
//=================================================
//=================================================
void getKinectIRImage (const sensor_msgs::ImageConstPtr&
   img);
void getKinectDepthImage (const sensor_msgs::
   ImageConstPtr& img);
void getUserData(const user_tracker::Com com);
bool detectFace();
void detectEyes();
void detectMovements();
CvPoint medianComputation(vector<Point2f> vector);
CvPoint medianComputation(vector<CvPoint> vector);
void drawArrow(pointer arrow, double hypotenuse, double
   angle, CvScalar color, int thickness);
bool isBlobIntoRect (Rect r, CvPoint c, unsigned int
   offset);
int defineAngleClass(double angle);
void userMotionCompensation(CvPoint pt1, CvPoint &pt2,
   double &angle);
void compileMessages(bool headDataMSG, bool EyeDataMSG,
   bool moveDataMSG);
float setEyeRatioThreshold(float ratio);
```

```
int getMinDistanceRect_y(vector<Rect> v, int point_y);

//Main
//=================================================
//=================================================
int main(int argc, char **argv)
{
 //Ros initialization
 ros::init(argc, argv, "head_analyzer");
 ros::NodeHandle nh;

 //Messages subscribers
 ros::Subscriber subIRKinect = nh.subscribe("camera/ir/
    image_raw", 1000, getKinectIRImage);
 ros::Subscriber subDepthKinect = nh.subscribe("camera/
    depth/image", 1000, getKinectDepthImage);
 ros::Subscriber subUserDistance = nh.subscribe("com",
    1000, getUserData);

 //Messages publishers
 pubHeadData = nh.advertise<head_analyzer::HeadDataMSG>("
    headData", 100);
 pubEyesData = nh.advertise<head_analyzer::EyesDataMSG>("
    eyesData", 100);
 pubMoveData = nh.advertise<head_analyzer::MoveDataMSG>("
    moveData", 100);

 //Initialization
 getFrame = true;
 IRImageReady = false;
 depthImageReady = false;
 userDistanceReady = false;
 detectMovesEnable = false;
 eyesFilteringEnable = true;

 userEyeLratio = 0.0;
 userEyeRratio = 0.0;
 userHeadRatio = 0.0;
 userHeadPitch = -1;
 userHeadRoll = -1;

 //move = new Array1D2DPointDataset();
 //movement = new Array1D2DPointDataset();
 knn = new KNNClassifier();
```

```
countHolesMove = 0;
moveIntensity = 0.0;

//Build/Load movesClassifier
ROS_INFO("[HEAD_ANALYZER]: Building Classifier...");
movesClassifier = ros::package::getPath("head_analyzer")
   +"/classifier/moveClassifier.dts";
knn->fastBuild(movesClassifier, 10, 5, 2);
ROS_INFO("[HEAD_ANALYZER]: Classifier Built");

//ROS LOOP
ros::Rate r(30);
while(ros::ok())
{
 //Messages control variables initialization
 headDataMessageReady = false;
 eyesDataMessageReady = false;
 moveDataMessageReady = false;

 if (IRImageReady && depthImageReady &&
    userDistanceReady && getFrame)
 {
  getFrame = false;

  cv::circle(frameDrawn, headPosition, 2, CV_RGB(0, 255,
      0), 3, 8, 0);

  if(userDistance <= USER_DISTANCE)
  {
   if(detectFace() && detectMovesEnable)
   {
    if(userDistance <= EYES_TRACKING_DISTANCE)
     detectEyes();

    detectMovements();
   }

   //Display analysis results
   //imshow("Head-Analyzer", frameDrawn);

   detectMovesEnable = true; //Enable starting from 2nd
      frame
  }

  frameIR.copyTo(prevFrameIR);
```

```
  }
  getFrame = true;

  if(waitKey(30) >= 0) break;

  //Send messages
  compileMessages(headDataMessageReady,
     eyesDataMessageReady,moveDataMessageReady);
  //ROS_INFO("Messaggi %d, %d, %d",headDataMessageReady,
     eyesDataMessageReady,moveDataMessageReady);

  ros::spinOnce();
  r.sleep();
 }

 return 0;
}

//Functions
//================================================
//================================================
float setEyeRatioThreshold(float ratio)
{
 if((ratio > 0.0) && (ratio <= 0.1)) { return 0.1; }
 else if((ratio > 0.1) && (ratio <= 0.2)) { return 0.2; }
 else if((ratio > 0.2) && (ratio <= 0.3)) { return 0.3; }
 else if((ratio > 0.3) && (ratio <= 0.4)) { return 0.4; }
 else if((ratio > 0.4) && (ratio <= 0.5)) { return 0.5; }
 else if((ratio > 0.5) && (ratio <= 0.6)) { return 0.6; }
 else if((ratio > 0.6) && (ratio <= 0.7)) { return 0.7; }
 else if((ratio > 0.7) && (ratio <= 0.8)) { return 0.8; }
 else if((ratio > 0.8) && (ratio <  1.0)) { return 1.0; }
 else
  { return -1.0; }
}

//================================================
//================================================
int getMinDistanceRect_y(vector<Rect> v, int point_y)
{
 int distance = 0;
 int max_distance = 65000;
 int idx;

 for (int i = 0; i < v.size(); i ++)
```

```
 {
  distance = fabs(point_y - (v[i].y + v[i].height/2));
  if(distance < max_distance)
  {
   max_distance = distance;
   idx = i;
  }
 }

 return idx;
}


//=================================================
//=================================================
void compileMessages(bool headDataMSG, bool eyeDataMSG,
   bool moveDataMSG)
{
 //Compiling head data message
 if(headDataMSG)
 {
  //set userHeadPitch threshold to send to e2_brain
  if (userHeadPitch < 0)
   { headData.headPitch.data = -1;}
  else if((userHeadPitch >= 0) && (userHeadPitch < 10))
   { headData.headPitch.data = 0; }
  else if((userHeadPitch >= 10) && (userHeadPitch < 20))
   { headData.headPitch.data = 10;}
  else if((userHeadPitch >= 20) && (userHeadPitch < 30))
   { headData.headPitch.data = 20; }
  else if((userHeadPitch >= 30) && (userHeadPitch < 40))
   { headData.headPitch.data = 30; }
  else if((userHeadPitch >= 40) && (userHeadPitch < 50))
   { headData.headPitch.data = 40; }
  else if((userHeadPitch >= 50) && (userHeadPitch < 60))
   { headData.headPitch.data = 50; }
  else if((userHeadPitch >= 60) && (userHeadPitch < 70))
   { headData.headPitch.data = 60; }
  else if((userHeadPitch >= 70) && (userHeadPitch < 80))
   { headData.headPitch.data = 70; }
  else if((userHeadPitch >= 80) && (userHeadPitch < 90))
   { headData.headPitch.data = 80; }

  if(userHeadRatio < 0.0)
   { headData.headRatio.data = -1.0; }
```

```
 else if((userHeadRatio >= 0.0) && (userHeadRatio < 0.1)
    )
  { headData.headRatio.data = 0.1; }
 else if((userHeadRatio >= 0.1) && (userHeadRatio < 0.2)
    )
  { headData.headRatio.data = 0.2; }
 else if((userHeadRatio >= 0.2) && (userHeadRatio < 0.3)
    )
  { headData.headRatio.data = 0.3; }
 else if((userHeadRatio >= 0.3) && (userHeadRatio < 0.4)
    )
  { headData.headRatio.data = 0.4; }
 else if((userHeadRatio >= 0.4) && (userHeadRatio < 0.5)
    )
  { headData.headRatio.data = 0.5; }
 else if((userHeadRatio >= 0.5) && (userHeadRatio < 0.6)
    )
  { headData.headRatio.data = 0.6; }
 else if((userHeadRatio >= 0.6) && (userHeadRatio < 0.7)
    )
  { headData.headRatio.data = 0.7; }
 else if((userHeadRatio >= 0.7) && (userHeadRatio < 0.8)
    )
  { headData.headRatio.data = 0.8; }
 else if((userHeadRatio >= 0.8) && (userHeadRatio < 0.9)
    )
  { headData.headRatio.data = 0.9; }

 if(userHeadRoll == -1)
  { headData.headRoll.data = -1; }
 else if((userHeadRoll >= -5) && (userHeadRoll <= 5))
  { headData.headRoll.data = 0; }
 else if(userHeadRoll < -5 || userHeadRoll > 5)
  { headData.headRoll.data = 1; }
}
else
{
 headData.headRatio.data = -1.0;
 headData.headPitch.data = -1;
 headData.headRoll.data = -1;
}

//Compiling eyesdata message
if(eyeDataMSG)
{
```

```
  eyesData.eyeLRatio.data = setEyeRatioThreshold(
     userEyeLratio);
  eyesData.eyeRRatio.data = setEyeRatioThreshold(
     userEyeRratio);
 }
 else
 {
  eyesData.eyeLRatio.data = -1.0;
  eyesData.eyeRRatio.data = -1.0;
 }

 //Compiling movedata message
 if(moveDataMSG)
 {
  moveData.moveClassification.data = userMoveClass;
 }
 else
  moveData.moveClassification.data = 'U';


 //Send messages
 pubHeadData.publish(headData);
 pubEyesData.publish(eyesData);
 pubMoveData.publish(moveData);

 //Print messages data (Debug)
 //ROS_INFO("Head Data:  Ratio: %f   --   Pitch: %d   --
      Roll: %d", headData.headRatio.data, headData.
    headPitch.data, headData.headRoll.data);
 //ROS_INFO("Eye  Data:  Left Ratio: %f   --  Right Ratio
    : %f", eyesData.eyeLRatio.data, eyesData.eyeRRatio.
    data);
}

//================================================
//================================================
void userMotionCompensation(CvPoint pt1, CvPoint &pt2,
   double &angle)
{
 //compute user movement
 CvPoint userMotion;
 userMotion.x = headPosition.x - headPositionPrev.x;
 ROS_INFO("userMotion.x : %d", userMotion.x);
 userMotion.y = headPosition.y - headPositionPrev.y;
```

```
 //Motion Compensation
 pt2.x = pt2.x - userMotion.x;
 pt2.y = pt2.y - userMotion.y;

 //Compute movement angle compensated
 angle = atan2((double)(pt1.y - pt2.y), (double)(pt1.x -
    pt2.x));

}


//=================================================
//=================================================
int defineAngleClass(double angle)
{
 if ((angle >= -0.3925) && (angle < 0.3925)) {return 3;}
 else if ((angle >= 0.3925) && (angle < 1.1775)) {return
    2;}
 else if ((angle >= 1.1775) && (angle < 1.9625)) {return
    1;}
 else if ((angle >= 1.9625) && (angle < 2.7475)) {return
    -2;}
 else if (((angle >= 2.7475) && (angle <= 3.14)) ||
      ((angle >= -3.14) && (angle < -2.7475))) {return
         -3;}
 else if ((angle >= -2.7475) && (angle < -1.9625)) {
    return -4;}
 else if ((angle >= -1.9625) && (angle < -1.1775)) {
    return -1;}
 else if ((angle >= -1.1774) && (angle < -0.3925)) {
    return 4;}

}


//=================================================
//=================================================
bool isBlobIntoRect (Rect r, CvPoint c, unsigned int
   offset)
{
 if (((c.x >= (r.x-offset)) && (c.x <= (r.x+r.width+
    offset))) &&
    ((c.y >= (r.y-offset)) && (c.y <= (r.y+r.height+
       offset))))
    return true;
```

```
 return false;
}


//================================================
//================================================
void drawArrow(pointer arrow, double hypotenuse, double
   angle, CvScalar color, int thickness)
{
 arrow.pt2.x = (int) (arrow.pt1.x - 3 * hypotenuse*cos(
    angle));
 arrow.pt2.y = (int) (arrow.pt1.y - 3 * hypotenuse*sin(
    angle));

/* if(arrow.pt2.x < 10 )
  arrow.pt2.x = 10;
 else if (arrow.pt2.x > FRAME_WIDTH-10)
  arrow.pt2.x = FRAME_WIDTH-10;
*/
 cv::line(frameDrawn, arrow.pt1, arrow.pt2, color,
    thickness, 8, 0);

 arrow.pt1.x = (int) (arrow.pt2.x + 9 * cos(angle + pi/4)
    );
 arrow.pt1.y = (int) (arrow.pt2.y + 9 * sin(angle + pi/4)
    );
 cv::line(frameDrawn, arrow.pt1, arrow.pt2, color,
    thickness, 8, 0);

 arrow.pt1.x = (int) (arrow.pt2.x + 9 * cos(angle - pi/4)
    );
 arrow.pt1.y = (int) (arrow.pt2.y + 9 * sin(angle - pi/4)
    );
 cv::line(frameDrawn, arrow.pt1, arrow.pt2, color,
    thickness, 8, 0);
}


//================================================
//================================================
CvPoint medianComputation(vector<Point2f> vector)
{
 CvPoint result;
```

```
int size = (int) vector.size();

multiset<int> vectorXcoord, vectorYcoord;
multiset<int>::iterator it;

//get x/y coordinate
for (int i=0; i< size; i++)
{
 vectorXcoord.insert(vector[i].x);
 vectorYcoord.insert(vector[i].y);
}

//median computation
if (size%2 != 0)  //odd number of elements
{
 it = vectorXcoord.begin();
 advance(it, ((size-1)/2) );
 result.x = *it;

 it = vectorYcoord.begin();
 advance(it, ((size-1)/2) );
 result.y = *it;
}
else       //even number of elements
{
 int temp1, temp2; //variable for saving the n/2_th and
    (n+1)/2_th elements

 it = vectorXcoord.begin();
 advance(it, ((size-1)/2) );
 temp1 = *it;
 it = vectorXcoord.begin();
 advance(it, (size/2) );
 temp2 = *it;
 result.x = (temp1+temp2)/2;

 it = vectorYcoord.begin();
 advance(it, ((size-1)/2) );
 temp1 = *it;
 it = vectorYcoord.begin();
 advance(it, (size/2) );
 temp2 = *it;
 result.y = (temp1+temp2)/2;
}
```

```
 return result;

}


//================================================
//================================================
CvPoint medianComputation(vector<CvPoint> vector)
{
 CvPoint result;

 int size = (int) vector.size();

 multiset<int> vectorXcoord, vectorYcoord;
 multiset<int>::iterator it;

 //get x/y coordinate
 for (int i=0; i< size; i++)
 {
  vectorXcoord.insert(vector[i].x);
  vectorYcoord.insert(vector[i].y);
 }

 //median computation
 if (size%2 != 0)  //odd number of elements
 {
  it = vectorXcoord.begin();
  advance(it, ((size-1)/2) );
  result.x = *it;

  it = vectorYcoord.begin();
  advance(it, ((size-1)/2) );
  result.y = *it;
 }
 else       //even number of elements
 {
  int temp1, temp2; //variable for saving the n/2_th and
     (n+1)/2_th elements

  it = vectorXcoord.begin();
  advance(it, ((size-1)/2) );
  temp1 = *it;
  it = vectorXcoord.begin();
  advance(it, (size/2) );
  temp2 = *it;
```

```
  result.x = (temp1+temp2)/2;

  it = vectorYcoord.begin();
  advance(it, ((size-1)/2) );
  temp1 = *it;
  it = vectorYcoord.begin();
  advance(it, (size/2) );
  temp2 = *it;
  result.y = (temp1+temp2)/2;
 }

 return result;

}


//================================================
//================================================
void detectMovements()
{
 Mat frameIR_1C, prevFrameIR_1C;

 //Convert prevFrameIR in a 1 channel image (
    prevFrameIR_1C)
 cvtColor(prevFrameIR, prevFrameIR_1C, CV_BGR2GRAY);

 //Convert frameIR in a 1 channel image (frameIR_1C)
 cvtColor(frameIR, frameIR_1C, CV_BGR2GRAY);

 //----- SHI & TOMASI ALGORITHM -----

 //Saving of prevFrameIR_1C features
 vector<Point2f> prevFrameIRFeatures;
 //mask setting
 Mat mask;
 CvPoint pt1Mask, pt2Mask;

 CvSize dim = cv::Size(frameIR.cols, frameIR.rows);
 mask = cv::Mat::zeros(dim, CV_8U);
 pt1Mask.x = eyesAreaBlob.getPt1().x + 10;
 pt1Mask.y = eyesAreaBlob.getPt1().y;
 pt2Mask.x = eyesAreaBlob.getPt2().x - 10;
 pt2Mask.y = eyesAreaBlob.getPt2().y;
```

```
cv::rectangle(mask, pt1Mask, pt2Mask, CV_RGB(255, 255,
    255), CV_FILLED, 8, 0);

//draw mask dimensions
//cv::rectangle(frameDrawn, pt1Mask, pt2Mask, CV_RGB(0,
    0, 255), 2, 8, 0);

//Computation of prevFrameIR_1C features
goodFeaturesToTrack(prevFrameIR_1C, prevFrameIRFeatures,
     FEATURES_NUMBER, .2, .1, mask, 3, false, 0.04);

//----- LUCAS AND KANADE ALGORITHM -----

//Saving of frameIR_1C features
vector<Point2f> frameIRFeatures;
vector<uchar> foundFeatures;
vector<float> featuresError;

TermCriteria terminationCriteria = TermCriteria(
    CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .03);

calcOpticalFlowPyrLK(prevFrameIR_1C, frameIR_1C,
       prevFrameIRFeatures, frameIRFeatures,
       foundFeatures, featuresError,
       cv::Size(15,15), 3, terminationCriteria, 0.5, 0);

//----- PRINT ALL GOOD FEATURES FOUND -----
/*for(int i=0; i< FEATURES_NUMBER; i++)
{
 pointer arrow;
 arrow.pt1.x = prevFrameIRFeatures[i].x;
 arrow.pt1.y = prevFrameIRFeatures[i].y;
 arrow.pt2.x = frameIRFeatures[i].x;
 arrow.pt2.y = frameIRFeatures[i].y;

 double angle = atan2((double) arrow.pt1.y - arrow.pt2.y
     , (double) arrow.pt1.x - arrow.pt2.x);
 double hypotenuse = sqrt(pow((arrow.pt1.y - arrow.pt2.y
    ),2) + pow((arrow.pt1.x - arrow.pt2.x),2));

 drawArrow(arrow, hypotenuse, angle, CV_RGB(0,255,0), 1)
     ;
}*/

//----- MEDIAN COMPUTATION -----
```

```
pointer median;
median.pt1 = medianComputation(prevFrameIRFeatures);
median.pt2 = medianComputation(frameIRFeatures);

//----- MOVEMENTS DATA COMPUTATION -----
double medianHypotenuse = sqrt(pow((median.pt1.y -
   median.pt2.y),2) + pow((median.pt1.x - median.pt2.x)
   ,2));
double medianAngle = atan2((double)(median.pt1.y -
   median.pt2.y), (double)(median.pt1.x - median.pt2.x)
   );

//----- PRINT MOVEMENT FOUND -----
pointer moveArrow;

//Print arrow upon the face
moveArrow.pt1.x = faceBlob.getPt1().x + (((faceBlob.
   getPt2().x)-(faceBlob.getPt1().x))/2);
moveArrow.pt1.y = faceBlob.getPt1().y;

//Considering only meaningful arrows
if(((unsigned int)medianHypotenuse > 10) && ((unsigned
   int)medianHypotenuse < 100))
{
 //reset count of holes during movements
 countHolesMove = 0;

 //draw arrow
 drawArrow(moveArrow, medianHypotenuse, medianAngle,
    CV_RGB(255,0,0), 2);

 //save hypotenuse  count
 moveIntensity = moveIntensity + medianHypotenuse;

 //save data of arrow
 //ROS_INFO("Angle: %f, AngleClass:%d", medianAngle,
    defineAngleClass(medianAngle));
 Point2DSample moveSample = Point2DSample(
    defineAngleClass(medianAngle), 1, 1.0, 'U');
 move.insert(moveSample);

}
else if(countHolesMove < MAX_NO_MOVES)
{
 countHolesMove++;
```

```
}
else
{
 vector<Point2DSample> interpolatedMoves;

 //pruning of meaningful recorded movements and
    interpolating of them
 if ((move.size() >= MIN_MOVES) && (move.size() <=
    MAX_MOVES))
 {
  moveIntensity = moveIntensity / move.size();

  //ROS_INFO(" --- MOVE DONE!!!---Intensity average: %f
     ", moveIntensity);
  interpolatedMoves = move.interpolate(
     INTERPOLATION_FACTOR, HERMITE_INTERPOLATION);
  //ROS_INFO("Ready to Classify A!");

  //saving of interpolated vector on a file
  for(int j = 0; j < INTERPOLATION_FACTOR; j++)
  {
   Point2DSample sample = Point2DSample(
      interpolatedMoves[j].getX(), j, 1.0, 'U');
   movement.insert(sample);
  }

  //ROS_INFO("Ready to Classify B!");

  //----ACTIVATE THIS PART FOR ENABLING CLASSIFICATION
     ----//
  vector<Point2DSample> temp = movement.getSamples();
  //Get movement classificatiuon using KNN Classifier
     with radius algorithm
  userMoveClass = knn->getClassification(temp, knnScores
     , knnClass, KNN_WITH_RADIUS, 2.0, WITHOUT_LOG);

  //Check of result based on intensity
  if(moveIntensity > DENYING_INTESITY && userMoveClass
     == 'D')
  {
   int temp, temp1;
   for (int i = 0; i < knnClass.size(); i++)
   {
    if(knnClass[i] == 'L')
     temp = knnScores[i];
```

```
      else if(knnClass[i] == 'R')
       temp1 = knnScores[i];

      if (temp >= temp1)
       userMoveClass = 'L';
      else
       userMoveClass = 'R';
     }
    }

    moveDataMessageReady = true;

    /*//----ACTIVATE THIS PART FOR CREATING DATASET ----//
    movement->save(movesClassifier, SAVE_AS_DOUBLE, true,
       true);*/

    //reset
    moveIntensity = 0.0;
    movement.clear();
   }
   move.clear();
 }
}


//=================================================
//=================================================
void detectEyes()
{
 //Variables
 Mat binaryFrame;
 Mat histBinaryFaceFrame;
 Mat contoursFrame;
 Mat temp1;

 vector<vector<Point> > contours;
 vector<Vec4i> hierarchy;

 vector<Rect> leftBlobs;
 vector<Rect> rightBlobs;
 vector<Rect> leftCandidatesEye;
 vector<Rect> rightCandidatesEye;

 Blob candidatedBlob;
 Rect aBlob;
```

```
Rect searchAreaForEyesFiltering;

unsigned int blobSize = 0;
float blobRatio = 0.0;
float blobsDistance = 0;
int xDiff = 0;
int yDiff = 0;

bool isLeft = false;
bool isRight = false;

//Convert IRImage from Kinect into grayScale image and
   cut eyesArea
cvtColor(frameIR, binaryFrame, CV_BGR2GRAY);

//Cut eyesBinaryFrame to obtain eyesArea image
Mat temp2 (binaryFrame, eyesAreaBlob.getRect());

//Distance handler
if (userDistance < 700)
{
 //Define blobs dimension
 MIN_EYE_BLOB_SIZE = 30;
 MAX_EYE_BLOB_SIZE = 300;

 //Get binary image and optimize it for blob analysis
 adaptiveThreshold (temp2, temp1, 255,
        ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY_INV, 89,
           0); //AirLab 125
 erode(temp1,contoursFrame, Mat());
}
else if ((userDistance >= 700)&&(userDistance < 760))
{
 //Define blobs dimension
 MIN_EYE_BLOB_SIZE = 40;
 MAX_EYE_BLOB_SIZE = 300;

 //Get binary image and optimize it for blob analysis
 adaptiveThreshold (temp2, temp1, 255,
        ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY_INV, 91,
           0); //AirLab 125
 erode(temp1,contoursFrame, Mat());

 //imshow("Binary Eyes Image", temp1);
 //imshow("Eroded Eyes Image", contoursFrame);
```

```
}
else
{
 //Define blobs dimension
 MIN_EYE_BLOB_SIZE = 35;
 MAX_EYE_BLOB_SIZE = 300;

 //Get binary image and optimize it for blob analysis
 adaptiveThreshold (temp2, temp1, 255,
      ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY_INV, 75, 0);
          //Airlab 111
 erode(temp1,contoursFrame, Mat());
}

//Find eyesBlob
findContours(contoursFrame, contours, hierarchy,
   CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE , eyesAreaBlob.
   getPt1());

//Filter contours and get the best ones
for(int i = 0; i >= 0 ; i = hierarchy[i][0] )
{
 if ((int)contours[i].size() > 4)
 {
  aBlob = boundingRect(contours[i]);
  if(eyesFilteringEnable)
  {
   //Filtering by blob dimensions
   blobSize = ((int)aBlob.width)*((int)aBlob.height);
   blobRatio = ((int)aBlob.width)/((int)aBlob.height);
   //Save blob into vector of candidated blobs
   candidatedBlob = Blob(cvPoint(aBlob.x, aBlob.y),
      aBlob.height, aBlob.width);

   if (((blobSize > MIN_EYE_BLOB_SIZE) && (blobSize <
      MAX_EYE_BLOB_SIZE)) && (blobRatio >
      BLOB_EYE_RATIO))
   {
    //Get distance between blob center and left/right
       edge of eyesAreaBlob
    unsigned int distDX = eyesAreaBlob.getPt2().x -
       candidatedBlob.getCenter().x;
    unsigned int distSX = candidatedBlob.getCenter().x -
        eyesAreaBlob.getPt1().x;
```

```
    //Put blobs into vector
    if( distDX >= distSX )  //SX
    {
     leftBlobs.push_back(aBlob);
    }
    else
    {
     rightBlobs.push_back(aBlob);
    }
   }
  }
 }
}

//LEFT BLOBS
if(leftBlobs.size() >= MAX_EYE_CANDIDATE_SIZE)
{
 for(int i = 0; i < MAX_EYE_CANDIDATE_SIZE; i ++)
 {
  int k = getMinDistanceRect_y(leftBlobs, eyesAreaBlob.
     getPt2().y);
  leftCandidatesEye.push_back(leftBlobs[k]);
  leftBlobs.erase(leftBlobs.begin() + k);
 }
}
else
{
 for(int i = 0; i < leftBlobs.size(); i ++)
 leftCandidatesEye.push_back(leftBlobs[i]);
}

//RIGHT BLOBS
if(rightBlobs.size() >= MAX_EYE_CANDIDATE_SIZE)
{
 for(int i = 0; i < MAX_EYE_CANDIDATE_SIZE; i ++)
 {
  int k = getMinDistanceRect_y(rightBlobs, eyesAreaBlob.
     getPt2().y);
  rightCandidatesEye.push_back(rightBlobs[k]);
  rightBlobs.erase(rightBlobs.begin() + k);
 }
}
else
{
 for(int i = 0; i < rightBlobs.size(); i ++)
```

```
  rightCandidatesEye.push_back(rightBlobs[i]);
}


//Draw all eyes candidates
for(int i = 0; i < leftCandidatesEye.size(); i ++)
 rectangle(frameDrawn, leftCandidatesEye[i], CV_RGB
    (255,0,0), 1,8,0);
for(int i = 0; i < rightCandidatesEye.size(); i ++)
 rectangle(frameDrawn, rightCandidatesEye[i], CV_RGB
    (0,255,0), 1,8,0);


/*
 *
 * Final filtering
 *
*/

if(leftCandidatesEye.size() == 1)
{
 isLeft = true;
 leftEye = leftCandidatesEye[0];
}
if(rightCandidatesEye.size() == 1)
{
 isRight = true;
 rightEye = rightCandidatesEye[0];
}

if(isLeft)
{
 //circle(frameDrawn, candidatedBlob.getCenter(), 2,
    CV_RGB(255,255,0), 2, 8, 0);
 rectangle(frameDrawn, leftEye, CV_RGB(0,0,255), 2,8,0);
 userEyeLratio = (float)leftEye.height / (float)leftEye.
    width;
 userHeadRoll = leftEye.y + leftEye.height/2;
}
else if(!isLeft)
 userEyeLratio = -1.0;

if(isRight)
{
```

```
  //circle(frameDrawn, candidatedBlob.getCenter(), 2,
     CV_RGB(255,255,0), 2, 8, 0);
  rectangle(frameDrawn, rightEye, CV_RGB(0,0,255), 2,8,0)
     ;
  userEyeRratio = (float)rightEye.height / (float)
     rightEye.width;
  userHeadRoll = userHeadRoll - (rightEye.y + rightEye.
     height/2);
 }
 else if(!isRight)
  userEyeRratio = -1.0;

 if(!isLeft || !isRight)
  userHeadRoll = 0;

 eyesDataMessageReady = true;
}

//================================================
//================================================
bool detectFace()
{
 //Variables
 int biggerContourIdx = 0;
 int contourArea = -1;

 Mat binaryFrame;
 Mat binaryFrameCopy;

 vector<vector<Point> > contours;
 vector<Vec4i> hierarchy;

 Rect headROI;
 Rect faceROI;
 Blob faceBlobempirical;
 Rect eyesROI;

 if(userDistance < 750)
 {
  //Face area into frameIR
  headROI = cvRect( (headPosition.x - BLOB_HEAD_WIDTH/2),
       //X
       headPosition.y,          //Y
       BLOB_HEAD_WIDTH, BLOB_HEAD_HEIGHT);
```

```
 //Convert IRImage into gray image and then into binary
    one
 cvtColor(frameIR, binaryFrame, CV_BGR2GRAY);
 binaryFrame = binaryFrame > THRESHOLD_HEAD_NEAR;
}
else
{
 //Face area into frameIR
 headROI = cvRect( headPosition.x - (BLOB_HEAD_WIDTH-10)
    /2,  //X
      headPosition.y,          //Y
      BLOB_HEAD_WIDTH-10, BLOB_HEAD_HEIGHT-20);
 //Convert IRImage into gray image and then into binary
    one
 cvtColor(frameIR, binaryFrame, CV_BGR2GRAY);
 binaryFrame = binaryFrame > THRESHOLD_HEAD_FAR;
}

//Chech out-of-frame error
if(headROI.x < 0)
 headROI.x = 0;
if(headROI.x > ( FRAME_WIDTH - headROI.width))
 headROI.x = FRAME_WIDTH - headROI.width;

if(headROI.y < 0)
 headROI.y = 0;
if(headROI.y > (FRAME_HEIGHT - headROI.height))
 headROI.y = FRAME_HEIGHT - (headROI.height+10);

//Define a sub-image for head detection algorithm
binaryFrame.copyTo(binaryFrameCopy);
Mat headBinaryFrame (binaryFrame, headROI);

//OpenCV find contours algorithm
findContours(headBinaryFrame, contours, hierarchy,
   CV_RETR_CCOMP,
   CV_CHAIN_APPROX_SIMPLE, cvPoint(headROI.x, headROI.y)
      );

//Filter contours and get the biggest one
for (int i = 0; i >= 0; i = hierarchy[i][0])
{
 //Draw contours
 //vScalar color = CV_RGB(rand()&255,rand()&255,rand()
    &255);
```

```
 //drawContours(frameDrawn, contours, i, color,
    CV_FILLED,8, hierarchy);

 headROI = boundingRect(contours[i]);

 //Get the biggest area
 int temp = headROI.width * headROI.height;
 if(temp > contourArea)
 {
  contourArea = temp;
  biggerContourIdx = i;
 }
}

//Save head dimensions
if(contourArea > 0)
{
 headROI = boundingRect(contours[biggerContourIdx]);
 headBlob = Blob(cvPoint(headROI.x, headROI.y), headROI.
    height, headROI.width);

 //imshow("BinaryFrame", binaryFrame);
 //rectangle(frameDrawn, headROI, CV_RGB(0,255,0), 2, 8,
     0);

 //Take some border around the image
 if(headBlob.getPt1().x < 150)
 {
  userDistanceReady = false;
  return false;
 }
 else if (headBlob.getPt2().x > 600)
 {
  userDistanceReady = false;
  return false;
 }
 if( headBlob.getPt1().y < 20)
 {
  userDistanceReady = false;
  return false;
 }
 else if(headBlob.getPt2().y > 360 )
 {
  userDistanceReady = false;
  return false;
```

```
}


//Define eyes area
eyesROI = cvRect(headROI.x, (headROI.y + headROI.height
   /8 + 15),
     headROI.width, 3*headROI.height/8);

//Shrink headROI width with findContours algorithm
   applied on eyesArea sub-image
//Define a sub-image for face detection algorithm
Mat faceBinaryFrame (binaryFrameCopy, eyesROI);

//Find face contours
contours.clear();
hierarchy.clear();
findContours(faceBinaryFrame, contours, hierarchy,
   CV_RETR_CCOMP,
   CV_CHAIN_APPROX_SIMPLE, cvPoint(eyesROI.x, eyesROI.y
      ));

//Filter contours and get the biggest one
biggerContourIdx = 0;
contourArea = -1;
for (int i = 0; i >= 0; i = hierarchy[i][0])
{
 faceROI = boundingRect(contours[i]);

 //Get the biggest area
 int temp = faceROI.width * faceROI.height;
 if(temp > contourArea)
 {
  contourArea = temp;
  biggerContourIdx = i;
 }
}

//Save face dimensions
if(contourArea > 0)
{
 faceROI = boundingRect(contours[biggerContourIdx]);
 faceBlob = Blob(cvPoint(faceROI.x, headROI.y), headROI
    .height, faceROI.width);
```

```
   //faceBlobempirical = Blob(cvPoint(faceROI.x, headROI.
      y), headROI.height, (headROI.height/4)*3);
   //rectangle(frameDrawn, faceBlobempirical.getRect(),
      CV_RGB(0,0,255), 2, 8, 0);

   eyesAreaBlob =  Blob( cvPoint((faceROI.x), (eyesROI.y
      -5)), //Pt1
          cvPoint((faceROI.x+faceROI.width),eyesROI.y+
             eyesROI.height)); //Pt2

   //Drawn face blob and eye area
   rectangle(frameDrawn, faceBlob.getRect(), CV_RGB
      (0,255,0), 2, 8, 0);
   rectangle(frameDrawn, eyesAreaBlob.getRect(), CV_RGB
      (255,0,0), 2, 8, 0);

   //Save ratio
   userHeadRatio = (float)faceBlob.getWidth() / (float)
      faceBlob.getHeight();
   if (userHeadRatio > 0.9) {userHeadRatio = -1.0; }

   userHeadPitch = faceBlob.getPt1().y - headPosition.y;
   headDataMessageReady = true;
   return true;
  }
 }
 return false;
}

//=================================================
//=================================================
void getUserData(const user_tracker::Com com)
{
    //Save user's distance from camera (z coord)
    userDistance = (int)com.comPoints.z;
    headPositionPrev = headPosition;
 headPosition.x = (int)com.headPoint.x;
 headPosition.y = (int)com.headPoint.y;

    //Check if there are still a user
    if (userDistance > 0 && ((headPosition.x > 10 &&
       headPosition.x < FRAME_WIDTH-10)))
        userDistanceReady = true;
    else
    {
```

```
        ROS_INFO("[HEAD_ANALYZER]::User out of camera");
        userDistanceReady = false;
    }
}

//=====================================================
//=====================================================
void getKinectDepthImage (const sensor_msgs::
   ImageConstPtr& img)
{
 cv_bridge::CvImagePtr cv_ptr;

 if (getFrame)
 {
  try
  {
   cv_ptr = cv_bridge::toCvCopy(img, "32FC1");
   cv_ptr->image.copyTo(frameDepth);

   depthImageReady = true;
  }
  catch (cv_bridge::Exception& e)
  {
   depthImageReady = false;

   ROS_ERROR("[HEAD-ANALYZER]::Error in capturing depth
      image");
   return;
  }
 }
}


//=====================================================
//=====================================================
void getKinectIRImage (const sensor_msgs::ImageConstPtr&
   img)
{

 if (getFrame)
 {
  frameIR = img_bridge_.imgMsgToCv(img, "bgr8");

  frameIR.copyTo(frameDrawn);
```

```
  IRImageReady = true;
 }
}
```

B.2 Il Processo d'Interazione: il modulo *e2-brain*

```
//================================================
// Authors: cristianmandelli@gmail.com
//          deborahzamponi@gmail.com
// Data: 11/02/2012
// Description: This is e2 brain. This module manages
   data from vision
// modules, combining them to control robot actions. This
    module uses
// other module named "modulename_api" that are servers
   that provides
// interaction primitives for speak, head expression,
   neck moves etc.
// They are called by interaction process.
// The interaction process is managed by mean of state
   machines.
//================================================
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <sstream>
#include <fstream>
//ROS
#include "ros/ros.h"
#include "ros/package.h"
//Actionlib
#include <speak_api/SpeakAction.h>
#include <head_api/HeadAction.h>
#include <neck_api/NeckAction.h>
#include <wheel_motor/WheelAction.h>
#include <kinect_motor/KinectAction.h>
#include <actionlib/client/simple_action_client.h>
//Modules
#include <user_tracker/Com.h>
#include <head_analyzer/MoveDataMSG.h>
#include <head_analyzer/HeadDataMSG.h>
#include <head_analyzer/EyesDataMSG.h>
#include <sonar/SonarData.h>
//OtherLibs
#include "StateMachine.h"

//Primitives API
#define PRIMITIVE_API 10
```

```
//User params
#define USER_Y_MIN_POSITION 150
#define USER_Y_MAX_POSITION 50
//Interaction params
#define INTERACTION_SAMPLES 60

using namespace std;

//Action clients definition
typedef actionlib::SimpleActionClient<speak_api::
   SpeakAction> SpeakClient;
typedef actionlib::SimpleActionClient<head_api::
   HeadAction> HeadClient;
typedef actionlib::SimpleActionClient<neck_api::
   NeckAction> NeckClient;
typedef actionlib::SimpleActionClient<wheel_motor::
   WheelAction> WheelClient;
typedef actionlib::SimpleActionClient<kinect_motor::
   KinectAction> KinectClient;

//Enum
enum transaction { INTERESTED, NOT_INTERESTED };
enum APIcodeDefinition { SPEAK = 0, HEAD = 2, NECK = 4};

struct apiCode
{
 int apiCode[PRIMITIVE_API];
 string apiPar[PRIMITIVE_API];
};

struct phraseData
{
 unsigned int id;
 string speed;
 string language;
 string data;
};

struct headStruct
{
 float ratio;
 int pitch;
 int roll;
};
```

```
struct eyesStruct
{
 float leftRatio;
 float rightRatio;
};

struct userStruct
{
 int distance;
 headStruct headData;
 eyesStruct eyesData;

 char movement;
};

struct vectorStruct
{
 vector<int> userDistanceVect;
 vector<float> userHeadRatioVect;
 vector<int> userHeadPitchVect;
 vector<int> userHeadRollVect;
 vector<float> userEyeLRatioVect;
 vector<float> userEyeRRatioVect;
 vector<char> userMovesVect;
};

//Global Scope
They are called
int err;
int userDistance;
int userDistanceThreshold;
int userDistanceY;
int userHeadPitch;
int userHeadRoll;
int interestedCounter = 0;
int not_interestedCounter = 0;
int sampleCounter = 0;

float userEyeLRatio;
float userEyeRRatio;
float userHeadRatio;

char userMoveClass;

bool userPositionDataReady;
```

```
bool userEyesDataReady;
bool userMoveDataReady;
bool userHeadRatioDataReady;
bool userHeadPitchDataReady;
bool userHeadRollDataReady;
bool sonarDataReady;
bool kinectMotorFree;
bool speakHandlerFree;
bool headHandlerFree;
bool neckHandlerFree;
bool visionDataCapture;
bool visionDataAnalyze;
bool firstInteraction;

vector<phraseData> phrases;
vector<int> sonarData;
userStruct userData;
apiCode activationAPI;

//Prototypes
//================================================
//================================================
//Server action done callbacks
void speakDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const speak_api::SpeakResultConstPtr& result)
               ;
void headDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const head_api::HeadResultConstPtr& result);
void neckDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const neck_api::NeckResultConstPtr& result);
void wheelDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const wheel_motor::WheelResultConstPtr&
               result);
void kinectDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const kinect_motor::KinectResultConstPtr&
               result);
//Server action activated callbacks
void speakActiveCallback();
void headActiveCallback();
void neckActiveCallback();
```

```
void wheelActiveCallback();
void kinectActiveCallback();
//Server action feedBack callback
void speakFeedbackCallback(const speak_api::
   SpeakFeedbackConstPtr& feed);
void headFeedbackCallback(const head_api::
   HeadFeedbackConstPtr& feed);
void neckFeedbackCallback(const neck_api::
   NeckFeedbackConstPtr& feed);
void wheelFeedbackCallback(const wheel_motor::
   WheelFeedbackConstPtr& feed);
void kinectFeedbackCallback(const kinect_motor::
   KinectFeedbackConstPtr& feed);
//Messages callbacks
void getUserPositionData(const user_tracker::Com com);
void getUserEyesData(const head_analyzer::EyesDataMSG
   eyes);
void getUserHeadData(const head_analyzer::HeadDataMSG
   head);
void getUserMoveData(const head_analyzer::MoveDataMSG
   move);
void getSonarData(const sonar::SonarData sonar);
//Behaviour functions
int getUserDistanceBehaviour(vector<int>* vector);
int getUserHeadRollBehaviour(vector<int>* roll, vector<
   float>* ratio);
int getUserHeadPitchBehaviour(vector<int>* vector);
int getUserLeftEyeRatioBehaviour(vector<float>* vector);
int getUserRightEyeRatioBehaviour(vector<float>* vector);
char detectMeaningfulMove(vector<char>* vector);
//Other functions
int getVectorOutputValue(vector<int>* vector);
float getVectorOutputValue(vector<float>* vector);
void loadSpeakConfigFile(string file);
void setStatusValue(int interested, int not_interested);
int setMaxValueBt(int a, int b);
void primitiveMSGParser(string primitiveStr, string
   paramsStr);

//Main
//=================================================
//=================================================
int main(int argc, char **argv)
{
 //Ros initialization
```

```
ros::init(argc, argv, "e2_brain");
ros::NodeHandle nh;

//Controls variables initializations
userPositionDataReady = false;
userEyesDataReady = false;
userMoveDataReady = false;
userHeadRatioDataReady = false;
userHeadPitchDataReady = false;
userHeadRollDataReady = false;
sonarDataReady = false;
kinectMotorFree = false;
speakHandlerFree = true;
headHandlerFree = true;
   neckHandlerFree = true;
visionDataCapture = false;
visionDataAnalyze = false;
firstInteraction = true;

//Variables
vectorStruct dataVectors; //Values from vision modules
Node* node;

//Messages subscribtions
   ros::Subscriber subUserDistance = nh.subscribe("com",
       10, getUserPositionData);
   ros::Subscriber subUserHeadData = nh.subscribe("
      headData", 10, getUserHeadData);
   ros::Subscriber subUserEyesData = nh.subscribe("
      eyesData", 10, getUserEyesData);
   ros::Subscriber subUserMoveData = nh.subscribe("
      moveData", 10, getUserMoveData);
   ros::Subscriber subSonarData = nh.subscribe("
      sonarData", 10, getSonarData);

//Load state machines that controls the interaction
   process
StateMachine *sm = new StateMachine();
string stateMachineConfigFile = ros::package::getPath("
   e2_brain")+"/config/state_machine_config/
   state_machine.txt";
err = sm->loadFromFile(stateMachineConfigFile);
sm->printMap();
if(err < 0)
```

```
 ROS_ERROR("[e2_brain]::Error␣loading␣state␣machine␣from
    ␣configuration␣file");
else
 ROS_INFO("[e2_brain]::State␣machine␣from␣configuration␣
    file␣...␣[OK]");

//Load speak configuration file
string speakConfigFile = ros::package::getPath("e2_brain
   ")+"/config/speak_config/speak.txt";
loadSpeakConfigFile(speakConfigFile);

//Clients definition
SpeakClient speakClient("speak_api", true);
HeadClient headClient("head_api", true);
NeckClient neckClient("neck_api", true);
WheelClient wheelClient("wheel_motor", true);
KinectClient kinectClient("kinect_motor", true);

//Servers goal definitions
speak_api::SpeakGoal speakGoal;
head_api::HeadGoal headGoal;
neck_api::NeckGoal neckGoal;
wheel_motor::WheelGoal wheelGoal;
kinect_motor::KinectGoal kinectGoal;

//Wait for servers
speakClient.waitForServer();
headClient.waitForServer();
neckClient.waitForServer();
wheelClient.waitForServer();
kinectClient.waitForServer();
ROS_INFO("[e2_brain]::Servers␣connected␣...␣[OK]");

//KinectMotor initializations
kinectGoal.tilt = -100;
kinectClient.sendGoal(kinectGoal, &kinectDoneCallback, &
   kinectActiveCallback, &kinectFeedbackCallback);
kinectMotorFree = false;

//Motor initializations
wheelGoal.rotSpeed = 0;
wheelGoal.tanSpeed = 0;
wheelClient.sendGoal(wheelGoal, &wheelDoneCallback, &
   wheelActiveCallback, &wheelFeedbackCallback);
```

```
//RosLoop
ros::Rate r(10);
while(ros::ok())
{
       //Kinect_motor management
 if(userPositionDataReady && userDistance < 1500)
 {
  if(userDistanceY > USER_Y_MIN_POSITION &&
     kinectMotorFree)
  {
   kinectMotorFree = false;
   kinectGoal.tilt = -2;
   kinectClient.sendGoal(kinectGoal, &kinectDoneCallback
       , &kinectActiveCallback, &kinectFeedbackCallback)
       ;
  }
  else if(userDistanceY < USER_Y_MAX_POSITION &&
     kinectMotorFree)
  {
   kinectMotorFree = false;
   kinectGoal.tilt = 2;
   kinectClient.sendGoal(kinectGoal, &kinectDoneCallback
       , &kinectActiveCallback, &kinectFeedbackCallback)
       ;
  }
 }

       //Vision Data Management
       if(userPositionDataReady && userDistance < 1000)
 visionDataCapture = true;

 if(visionDataCapture && !visionDataAnalyze)
 {
  if(firstInteraction)
  {
   sampleCounter = 59;
  }
  //ROS_INFO("VISION DATA %d", sampleCounter);
  //Get user position data
  if(userPositionDataReady)
  {
   dataVectors.userDistanceVect.push_back(
      userDistanceThreshold);
   userPositionDataReady = false;
   sampleCounter++;
```

```
    }
    //Get user head data
    if(userHeadPitchDataReady)
    {
     dataVectors.userHeadPitchVect.push_back(userHeadPitch
        );
     userHeadPitchDataReady = false;
    }
    if(userHeadRatioDataReady)
    {
     dataVectors.userHeadRatioVect.push_back(userHeadRatio
        );
     userHeadRatioDataReady = false;
    }
    if(userHeadRollDataReady)
    {
     dataVectors.userHeadRollVect.push_back(userHeadRoll);
     userHeadRollDataReady = false;
    }
    //Get user eyes data
    if(userEyesDataReady)
    {
     dataVectors.userEyeLRatioVect.push_back(userEyeLRatio
        );
     dataVectors.userEyeRRatioVect.push_back(userEyeRRatio
        );
     userEyesDataReady = false;
    }
    //Get user move data
    if(userMoveDataReady)
    {
     dataVectors.userMovesVect.push_back(userMoveClass);
     userMoveDataReady = false;
    }

    //Stop acquisition and start analysis when all data
       are acquired
    if((sampleCounter == INTERACTION_SAMPLES) && (
       speakHandlerFree))
    {
     //ROS_INFO("Interaction Sample acquired");
     //ROS_INFO(" Samples details:");
     //ROS_INFO("      UserTracker:  %d", dataVectors.
        userDistanceVect.size());
```

```
    //ROS_INFO("      HeadAnalyzer: %d", dataVectors.
       userHeadRatioVect.size());
    visionDataCapture = false;
    visionDataAnalyze = true;
   }
  }

        //Vision data analysis
  if(visionDataAnalyze)
  {
   int t = 1;
   if(!firstInteraction)
   {
    //User Distance analysis
    userData.distance = getUserDistanceBehaviour(&
       dataVectors.userDistanceVect);
    ROS_ERROR("userDistanceVector -> [OK]");
    if (userData.distance > 0) {setStatusValue(-1, 1);}
    else if (userData.distance = 0) {setStatusValue(1, 0)
       ;}
    else if (userData.distance < 0) {setStatusValue(2,
       -2);}

    //User Ratio and Roll analysis
    userData.headData.roll = getUserHeadRollBehaviour(&
       dataVectors.userHeadRollVect, &dataVectors.
       userHeadRatioVect);
    ROS_ERROR("userHeadRollVector -> [OK]");
    userData.headData.ratio = userData.headData.roll;
    ROS_ERROR("userHeadRatioVector -> [OK]");
    if (userData.headData.roll != 0) {setStatusValue(2,
       -2);}
    else if (userData.headData.roll = 0) {setStatusValue
       (1, 0);}

    //User Pitch analysis
    userData.headData.pitch = getUserHeadPitchBehaviour(&
       dataVectors.userHeadPitchVect);
    ROS_ERROR("userHeadPitchVector -> [OK]");
    if (userData.headData.pitch > 0) {setStatusValue(-2,
       2); }
    else if (userData.headData.pitch = 0) {setStatusValue
       (1, 1); }
    else if (userData.headData.pitch < 0) {setStatusValue
       (2, -2); }
```

```
//User left eye ratio analysis
userData.eyesData.leftRatio =
    getUserLeftEyeRatioBehaviour(&dataVectors.
    userEyeLRatioVect);
ROS_ERROR("userleftEyeRatioVector -> [OK]");
if (userData.eyesData.leftRatio > 0) {setStatusValue
    (1, -1); }
else if (userData.eyesData.leftRatio = 0) {
    setStatusValue(0, 0); }
else if (userData.eyesData.leftRatio < 0) {
    setStatusValue(-1, 1); }

//User right eye ratio analysis
userData.eyesData.rightRatio =
    getUserRightEyeRatioBehaviour(&dataVectors.
    userEyeRRatioVect);
ROS_ERROR("userRightEyeRatioVector -> [OK]");
if (userData.eyesData.rightRatio > 0) {setStatusValue
    (1, -1); }
else if (userData.eyesData.rightRatio = 0) {
    setStatusValue(0, 0); }
else if (userData.eyesData.rightRatio < 0) {
    setStatusValue(-1, 1); }

//User move analysis
userData.movement = detectMeaningfulMove(&dataVectors
    .userMovesVect);
ROS_INFO("Detected moves: %c", userData.movement);
if (userData.movement == 'N') {setStatusValue(3, -2);
     }
else if (userData.movement == 'D') {setStatusValue
    (-2, 3); }
else if (userData.movement == 'L') {setStatusValue
    (-1, 2); }
else if (userData.movement == 'R') {setStatusValue
    (-1, 2); }

ROS_ERROR("Counters value INTERESTED=%d
    NOT_INTERESTED=%d",
interestedCounter, not_interestedCounter);

//Select interaction action
t = setMaxValueBt(interestedCounter,
    not_interestedCounter);
```

```
}
//Clear vector
dataVectors.userDistanceVect.clear();
dataVectors.userHeadRatioVect.clear();
dataVectors.userHeadPitchVect.clear();
dataVectors.userHeadRollVect.clear();
dataVectors.userEyeLRatioVect.clear();
dataVectors.userEyeRRatioVect.clear();
dataVectors.userMovesVect.clear();
interestedCounter = 0;
not_interestedCounter = 0;

firstInteraction = false;

//run one step into interaction procesess
if (t >= 0)
{
 ROS_ERROR("Run␣Machine␣with␣%d", t);
 node = sm->run(t);
 if(node != NULL)
 {
  //Print node info
  node->displayNode();

  //Check state machines output and send goals to
     servers
  primitiveMSGParser(node->getOutput(), node->
     getParams());

  /*for(int i=0; i < 10;  i++)
  {
   ROS_ERROR("code %d", activationAPI.apiCode[i]);
   ROS_ERROR("param %s", activationAPI.apiPar[i].c_str
      ());

  }*/


  //Enable capturing fase
  visionDataCapture = true;
  visionDataAnalyze = false;
  sampleCounter = 0;
 }
 else
```

```
   {
    visionDataCapture = false;
    visionDataAnalyze = false;
    ROS_ERROR("␣#######␣INTERAZIONE␣COMPLETATA␣#######␣"
        );
   }
  }
}

      //Send data to server (ActionLib)
//Speaking
if(activationAPI.apiCode[SPEAK] == 1)
{
 activationAPI.apiCode[SPEAK] = -1;
 speakHandlerFree = false;

 //Get phrase number
 stringstream ss(activationAPI.apiPar[SPEAK]);
 int temp;
 ss >> temp;

 //Define goal and send it to the server side
 speakGoal.speed = phrases[temp].speed;
 speakGoal.language = phrases[temp].language;
 speakGoal.phrase = phrases[temp].data;
 speakClient.sendGoal(speakGoal, &speakDoneCallback, &
     speakActiveCallback, &speakFeedbackCallback);
}

//Head expression
if(activationAPI.apiCode[HEAD] == 1)
{
 activationAPI.apiCode[HEAD] = -1;
 headHandlerFree = false;
 int temp;
 //Get head action code (first value of params)
 size_t found = 0;
 found = activationAPI.apiPar[HEAD].find_first_of(",");
 if(found > 0)
 {
  string activationCode = activationAPI.apiPar[HEAD].
      substr(0, found);
  stringstream ss(activationCode);
  ss >> temp;
  //Remove first params to string
```

```
   activationAPI.apiPar[HEAD].erase(0, found + 1);

   //Set goal gode
   headGoal.actionCode = temp;
  }
  else
  {
   stringstream ss(activationAPI.apiPar[HEAD]);
   ss >> temp;
   //Set goal gode
   headGoal.actionCode = temp;
  }

  //Define goal and send it to the server side
  headGoal.params = activationAPI.apiPar[HEAD];
  headClient.sendGoal(headGoal, &headDoneCallback, &
     headActiveCallback, &headFeedbackCallback);
 }

 //Neck moves
 if(activationAPI.apiCode[NECK] == 1)
 {
  activationAPI.apiCode[NECK] = -1;

  neckHandlerFree = false;
  int temp;
  //Get head action code (first value of params)
  size_t found = 0;
  found = activationAPI.apiPar[NECK].find_first_of(",");
  if(found > 0)
  {
   string activationCode = activationAPI.apiPar[NECK].
      substr(0, found);
   stringstream ss(activationCode);
   ss >>temp;
   //Remove fiurst params to string
   activationAPI.apiPar[NECK].erase(0, found + 1);

   //Set goal gode
   neckGoal.actionCode = temp;
  }
  else
  {
   stringstream ss(activationAPI.apiPar[NECK]);
   ss >> temp;
```

```
    //Set goal code
    neckGoal.actionCode = temp;
   }

   //Define goal and send it to the server side
   neckGoal.params = activationAPI.apiPar[NECK];
   neckClient.sendGoal(neckGoal, &neckDoneCallback, &
      neckActiveCallback, &neckFeedbackCallback);

  }

        //Ending i-th iteration
  ros::spinOnce();
  r.sleep();
 }
}
//Functions
//=================================================
//=================================================
void primitiveMSGParser(string primitiveStr, string
   paramsStr)
{

 unsigned int it = 0;
 size_t found;
 string functionParams, functionPar;

 for(size_t i = 0; i < primitiveStr.length(); i++)
 {
  if(primitiveStr.at(i) == '1')
  {
   //Set apiCode activation on true value
   activationAPI.apiCode[i] = 1;
   //find first occurence of ":"
   found = paramsStr.find_first_of(":");
   //save the function params into activation api struct
   activationAPI.apiPar[i] = paramsStr.substr(0, found);
   //erase the substring from starting string
   paramsStr.erase(0, found+1);
  }
  else
  {
   activationAPI.apiCode[i] = -1;
   activationAPI.apiPar[i] = "␣";
  }
```

```
  it++;
 }
}

//================================================
//================================================
void getSonarData(const sonar::SonarData sonar)
{
 //Clear previous data
 sonarData.clear();

 //Read received data
 sonarData.push_back(sonar.sonarA.data);
 sonarData.push_back(sonar.sonarB.data);
 sonarData.push_back(sonar.sonarC.data);
 sonarData.push_back(sonar.sonarD.data);
 sonarData.push_back(sonar.sonarE.data);
 sonarData.push_back(sonar.sonarF.data);
 sonarData.push_back(sonar.sonarG.data);
 sonarData.push_back(sonar.sonarH.data);

 sonarDataReady = true;

 //for(int i = 0; i < sonarData.size(); i ++)
 // ROS_INFO("[e2_brain]::SonarData received %d",
    sonarData[i]);
}

//================================================
//================================================
void getUserEyesData(const head_analyzer::EyesDataMSG
   eyesData)
{
 userEyeLRatio = (float)eyesData.eyeLRatio.data;
 userEyeRRatio = (float)eyesData.eyeRRatio.data;

 if(userEyeLRatio > 0.0 && userEyeRRatio > 0.0)
  userEyesDataReady = true;
 else
  userEyesDataReady = false;

}


//================================================
```

```
//================================================
void getUserHeadData(const head_analyzer::HeadDataMSG
   headData)
{
 userHeadRatio = (float)headData.headRatio.data;
 if( userHeadRatio > 0.0 )
  userHeadRatioDataReady = true;
 else
  userHeadRatioDataReady = false;

 userHeadPitch = (int)headData.headPitch.data;
 if( userHeadPitch > 0 )
  userHeadPitchDataReady = true;
 else
  userHeadPitchDataReady = false;

 userHeadRoll = (int)headData.headRoll.data;
 if( userHeadRoll != -1 )
  userHeadRollDataReady = true;
 else
  userHeadRollDataReady = false;

}


//================================================
//================================================
void getUserMoveData(const head_analyzer::MoveDataMSG
   moveData)
{
 userMoveClass = (char)(moveData.moveClassification.data)
    .at(0);
 if(userMoveClass != 'U')
  userMoveDataReady = true;

 //ROS_INFO("[e2_brain]::UserMoveDataMSG received = %c ",
     userMoveClass);
}


//================================================
//================================================
void getUserPositionData(const user_tracker::Com com)
{
    //Save user's distance from camera (z coord)
```

```
    userDistance = (int)com.comPoints.z;
    userDistanceThreshold = (int)com.distanceThreshold.
       data;
    userDistanceY = (int)com.headPoint.y;

    //Check if there are still a user
    if (userDistance > 0)
    {
      userPositionDataReady = true;
  }
    else
    {
        ROS_INFO("User out of camera");
        userPositionDataReady = false;
    }
    //ROS_INFO("[e2_brain]::UserPositionDataMSG received
       = Distance %d  --  DistantThr %d", userDistance,
       userDistanceThreshold);
}


//=================================================
//=================================================
void loadSpeakConfigFile(string file)
{
 phraseData temp;
 string fileLine;
 size_t found;
 int i = 0;

 //Open file
 ifstream File;
 File.open(file.c_str(), ios::in);

 while(!File.eof())
 {
  getline(File, fileLine);
  if (fileLine.length() > 0)
  {
   //find first occurence of ":"
   found = fileLine.find_first_of(":");
   if(found == 0) ROS_ERROR("[e2_brain]::Error parsing
      speak configuration file");

   //Get speed param
```

```
    temp.speed = fileLine.substr(0,found);
    fileLine.erase(0, found+1);

    found = fileLine.find_first_of(":");
    if(found == 0) ROS_ERROR("[e2_brain]::Error␣parsing␣
        speak␣configuration␣file");

    //Get language param
    temp.language = fileLine.substr(0,found);
    fileLine.erase(0, found+2);

    fileLine.erase(fileLine.length()-1,fileLine.length());
    temp.data = fileLine;
    temp.id = (unsigned int)i;

    phrases.push_back(temp);
    i++;
   }
  }

  ROS_INFO("[e2_brain]::Speak␣configuration␣file␣...␣[OK]"
      );
  File.close();
}


//================================================
//================================================
int setMaxValueBt(int a, int b)
{
  pair <int, int> values[2];
  pair <int, int> highest;

  //initialization highest
  highest.first = -1;
  highest.second = -100;

  //save values into array
  values[0].first = INTERESTED ; values[0].second = a ;
  values[1].first = NOT_INTERESTED ; values[1].second =  b
      ;

  //detect maximum value
  for(int i = 0; i < 2; i++)
  {
```

```
  //ROS_INFO("Current value: %d => %d", values[i].first,
     values[i].second);
  if(values[i].second > highest.second)
  {
   highest.first = values[i].first;
   highest.second = values[i].second;
   //ROS_INFO("Highest: %d => %d", highest.first, highest
      .second);
  }
  else if (values[i].second == highest.second)
  {
   //checking of equal value
   if(((values[i].first == INTERESTED) && (highest.first
      == NOT_INTERESTED)) ||
     ((values[i].first == NOT_INTERESTED) && (highest.
        first == INTERESTED)))
    {
     if ((userData.movement == 'D') || (userData.movement
         == 'R') ||
       (userData.movement == 'L'))
      { highest.first = NOT_INTERESTED; }
     else { highest.first = INTERESTED; }
    }
  }
 }

 ROS_INFO("Chosen Highest: %d => %d", highest.first,
    highest.second);
 return highest.first;

}

//================================================
//================================================
char detectMeaningfulMove(vector<char>* vector)
{
 map<char, int> counter;
 map<char,int>::iterator it;

 pair<char,int> highest;
 highest.first = 'U';
 highest.second = 0;

 //put vector value into map
 for(int i = 0; i < vector->size(); i++)
```

```
 {
  if(counter.count(vector->at(i))>0)
  {
   it = counter.find(vector->at(i));
   (*it).second += 1;
  }
  else
  {
    counter.insert(pair<char,int>(vector->at(i),1));
  }
 }

 //go through map for finding max value
 for(it=counter.begin() ; it != counter.end(); it++)
 {
  //cout<<(*it).first<<" => "<<(*it).second<<endl;
  //define max value
  if((*it).second > highest.second)
  {
   highest = make_pair((*it).first, (*it).second);
   cout<<"Highest: "<<highest.first<<" => "<<highest.
      second<<endl;
  }
  else if((*it).second == highest.second)
  {
   highest.first = 'U';
  }
 }

 return highest.first;
}


//=================================================
//=================================================
int getUserRightEyeRatioBehaviour(vector<float>*
   rightEyeRatio)
{
 int result = 0;

 result = getVectorOutputValue(rightEyeRatio);

 return result;
}
```

```
//=================================================
//=================================================
int getUserLeftEyeRatioBehaviour(vector<float>*
   leftEyeRatio)
{
 int result = 0;

 result = getVectorOutputValue(leftEyeRatio);

 return result;
}


//=================================================
//=================================================
int getUserHeadPitchBehaviour(vector<int>* pitch)
{
 int result = 0;

 //Detect biggest behaviour inside vector
 result = getVectorOutputValue(pitch);

 return result;
}


//=================================================
//=================================================
int getUserHeadRollBehaviour(vector<int>* roll, vector<
   float>* ratio)
{
 int result = 0;

 //Detect biggest behaviour inside vector
 result = getVectorOutputValue(roll);

 return result;
}


//=================================================
//=================================================
int getUserDistanceBehaviour(vector<int>* distance)
{
```

```
 int result = 0;

 //Detect biggest behaviour inside vector
 result = getVectorOutputValue(distance);

 return result;
}

//=================================================
//=================================================
float getVectorOutputValue(vector<float>* vector)
{
 float result = 0.0;
 float temp = vector->at(0);

 for(int i = 1; i < vector->size(); i++)
 {
  if (vector->at(i) > temp) { result++; }
  else if ((vector->at(i) < temp)) { result--; }

  temp = vector->at(i);
 }

 return result;
}


//=================================================
//=================================================
void setStatusValue(int interested, int not_interested)
{
 interestedCounter =  interestedCounter + interested;
 not_interestedCounter =  not_interestedCounter +
    not_interested;

}


//=================================================
//=================================================
int getVectorOutputValue(vector<int>* vector)
{
 int result = 0;
 int temp = vector->at(0);
```

```
 for(int i = 1; i < vector->size(); i++)
 {
  if (vector->at(i) > temp) { result++; }
  else if ((vector->at(i) < temp)) { result--; }

  temp = vector->at(i);
 }

 return result;
}

//=====================================================
//=====================================================
void speakDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const speak_api::SpeakResultConstPtr& result)
{
 ROS_INFO("[e2_brain]::Speak␣done␣[exit␣status␣%d]", (int
    )result->status);
 sampleCounter = 40;
 speakHandlerFree = true;
}

//=====================================================
//=====================================================
void speakActiveCallback()
{

}


//=====================================================
//=====================================================
void speakFeedbackCallback(const speak_api::
   SpeakFeedbackConstPtr& feed)
{

}


//=====================================================
//=====================================================
void headDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const head_api::HeadResultConstPtr& result)
```

```
{
 ROS_INFO("[e2_brain]::head␣moves␣done␣[exit␣status␣%d]",
     (int)result->status);
 headHandlerFree = true;
}

//================================================
//================================================
void headActiveCallback()
{}


//================================================
//================================================
void headFeedbackCallback(const head_api::
   HeadFeedbackConstPtr& feed)
{}


//================================================
//================================================
void neckDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const neck_api::NeckResultConstPtr& result)
{
 ROS_INFO("[e2_brain]::neck␣moves␣done␣[exit␣status␣%d]",
     (int)result->status);
 neckHandlerFree = true;
}

//================================================
//================================================
void neckActiveCallback()
{}


//================================================
//================================================
void neckFeedbackCallback(const neck_api::
   NeckFeedbackConstPtr& feed)
{}


//================================================
//================================================
```

```
void wheelDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const wheel_motor::WheelResultConstPtr&
               result)
{
 ROS_INFO("[e2_brain]::Motor␣Wheel␣set␣point␣reached␣[
    exit␣status␣%d]", (int)result->status);
}

//=================================================
//=================================================
void wheelActiveCallback()
{}


//=================================================
//=================================================
void wheelFeedbackCallback(const wheel_motor::
   WheelFeedbackConstPtr& feed)
{}


//=================================================
//=================================================
void kinectDoneCallback(const actionlib::
   SimpleClientGoalState& state,
            const kinect_motor::KinectResultConstPtr&
               result)
{
 //ROS_INFO("[e2_brain]::Kinect motor set point reached [
    exit status %d]", (int)result->status);
 kinectMotorFree = true;
}

//=================================================
//=================================================
void kinectActiveCallback()
{}


//=================================================
//=================================================
void kinectFeedbackCallback(const kinect_motor::
   KinectFeedbackConstPtr& feed)
{}
```