MAIN

```
/*
 * 2019/10/02 - Nicola Ariutti
 * Using 4 MPR121 to detect proximity on 25 Pads
 * Also Using a NeoPixel LED strip.
 */

/* CAPACITIVE STUFF ****************************************************************/
#include <Wire.h>
#include "Limulo_MPR121.h"

#define FIRST_MPR_ADDR 0x5A
const int  NMPR = 4;
const int  NPADS[] = {7, 6, 6, 6};
// virtual pads number will be calculated inside setup function
int NVIRTUALPADS = 0;
int padIndex, virtualPadIndex;

struct mpr121
{
  Limulo_MPR121 cap;
  uint8_t addr;
  // Save an history of the pads that have been touched/released
  uint16_t lasttouched = 0;
  uint16_t currtouched = 0;
  boolean bPadState[12];
  uint16_t oor=0;
};

// an array of mpr121! You can have up to 4 on the same i2c bus
mpr121 mpr[NMPR];

// utility variables to save values readed from MPR
uint16_t filt;
uint16_t base;
byte b;
int oor;

/* TOUCH LOGIC STUFF ***************************************************************/
#include "Touch.h"
Touch touchObj;
int oldIdx = -1;
int newIdx = -1;
// wait some time before sending data to VVVV
// in order stabilize the touch.
unsigned long waitToSend = 10;
unsigned long startWaiting;
boolean bTouchStabilized = false;


/* LED STUFF ***********************************************************************/
// TODO: change here to adapt the code for NeoPixel LED Strip
// this is the order of the colors GREEN, RED, BLUE
#include "Adafruit_NeoPixel.h"

#define NLEDS 68 // Number of LEDs in strip

// Here's how to control the LEDs from any two pins:
#define LEDPIN    4
Adafruit_NeoPixel strip = Adafruit_NeoPixel(NLEDS, LEDPIN, NEO_RGBW + NEO_KHZ800);

#include "Carriage.h"
Carriage carriage;

/* DEBUG ***************************************************************************/
```

```
boolean bToPlotter = false; // this boolean is set by Processing Plotter
boolean b2VVVV     = false;
boolean b2SC       = false; // when you want to have a SuperCollider sound feedback
boolean DEBUG      = false;
boolean bUseProcessing = false;
boolean bUseLEDs   = true;

/* UTILITY STUFF *******************************************************************/
int i=0, j=0;
const int DELAY_TIME = 10;

// SETUP ////////////////////////////////////////////////////////////////////////////

void setup()
{
  pinMode(13, OUTPUT); digitalWrite(13, LOW); // turn off the annoying LED

  // open the serial communication
  Serial.begin(9600, SERIAL_8N1);
  while(!Serial) {}

  if( DEBUG ) Serial.println("Starting!");

  int TOTALNUMBERPADS = 0;
  for(int i=0; i<NMPR; i++) {
    TOTALNUMBERPADS += NPADS[i];
  }
  NVIRTUALPADS = (TOTALNUMBERPADS * 2) - 1;

  if(DEBUG) {
    Serial.print("NMPR: ");
    Serial.print( NMPR );
    Serial.print(", NPADS tot: ");
    Serial.print( TOTALNUMBERPADS );
    Serial.print(", NVIRTUALPADS: ");
    Serial.print(NVIRTUALPADS);
    Serial.println(";");
  }

  // CAPACITIVE STUFF **************************************************************/
  // cycle through all the MPR
  for(int i=0; i<NMPR; i++)
  {
    mpr[i].cap = Limulo_MPR121();
    // mpr address can be one of these: 0x5A, 0x5B, 0x5C o 0x5D
    mpr[i].addr = FIRST_MPR_ADDR + i;

    // Look for the MPR121 chip on the I2C bus
    if(DEBUG) Serial.println("Looking for MPRs!");
    if ( !mpr[i].cap.begin( mpr[i].addr ) )
    {
      if(DEBUG) Serial.println("MPR121 not found, check wiring?");
      while(1);
    }
    if(DEBUG) Serial.println("MPR found!");

    // initialize the array of booleans
    for(int j=0; j<12; j++) {
      mpr[i].bPadState[j] = false;
    }

    // possibly initialize the mpr with some initial settings
    mpr[i].cap.setUSL(201);
    mpr[i].cap.setTL(180);
    mpr[i].cap.setLSL(130);
    // First Filter Iteration
    // Second Filter Iteration
    // Electrode Sample Interval
    // NOTE: the system seems to behave better if
    // these value are more than 0
    mpr[i].cap.setFFI_SFI_ESI(1, 1, 1);  // See AN3890

    // TODO: 2019-10-02
    // use some trick from FLOS capacitive wall here

    // MHD, NHD, NCL, FDL
    mpr[i].cap.setFalling( 4, 4, 2, 1 ); // 1, 1, 1, 1
    mpr[i].cap.setRising( 1, 1, 1, 1 );
    // if touch timing is too short there will be a moment in which
    // an automatic release will be interpreted as a user release
    // so a change in the pad index :(
    mpr[i].cap.setTouched( 1, 1, 32 );
    mpr[i].cap.setThresholds( 25, 5 ); // originariamente: 18, 9
    mpr[i].cap.setDebounces(2, 2);

    // initial reset of MPR struct fields
  }

  // LED STUFF *******************************************************************/
  //if(bUseLEDs) carriage.init(&strip, NVIRTUALPADS);
  if(bUseLEDs) carriage.init(&strip, 25);
}


// LOOP ////////////////////////////////////////////////////////////////////////////

void loop()
{
  // get data from serial port
  getSerialData();

  // CAPACITIVE STUFF *************************************************************/
  for(int i=0; i<NMPR; i++)  // cycle through all the MPR
  {
    // Get the currently touched pads
    mpr[i].currtouched = mpr[i].cap.touched();

    if( mpr[i].currtouched != mpr[i].lasttouched )
    {
      // do the following only if something has changed
      //Serial.println(mpr[i].currtouched, BIN);

      // interpolated Pad is the virtual indexes we obtain from the 'Touch obj'.
      virtualPadIndex = -1;
      padIndex = -1;
      for(int j=0; j<NPADS[i]; j++) // cycle through all the electrodes
      {
        if (( mpr[i].currtouched & _BV(j)) && !(mpr[i].lasttouched & _BV(j)) )
        {
          // pad 'j' has been touched
          mpr[i].bPadState[j] = true;

          padIndex = composeIndex(i, j);
          touchObj.addIndex( padIndex );
          virtualPadIndex = touchObj.getInterpolatedIndex();

          newIdx = virtualPadIndex;
          //newIdx = padIndex;
          bTouchStabilized = false;

          if( bUseProcessing )
            printAllSensors();

          if( DEBUG ) {
            Serial.print("P - "); // P for pressed
            printAllSensors();
            Serial.print("REAL: ");
            Serial.print( padIndex );
            Serial.print(", VIRTUAL: ");
            Serial.print( virtualPadIndex );
            Serial.println();
          }

          if( b2VVVV ) {
            sentToVVVV( virtualPadIndex );
          }

          if( b2SC && virtualPadIndex!=-1) {
            Serial.print( virtualPadIndex );
            Serial.print('a');
            //Serial.println();
          }

          if(bUseLEDs) {
            //carriage.setNewPos( virtualPadIndex );
            carriage.setNewPos( newIdx );
            if( DEBUG ) {
              carriage.debug();
            }
          }
        }
        else if (!(mpr[i].currtouched & _BV(j)) && (mpr[i].lasttouched & _BV(j)) )
        {
          // pad 'i' has been released
          mpr[i].bPadState[j] = false;

          padIndex = composeIndex(i, j);

          touchObj.removeIndex( padIndex );
          virtualPadIndex = touchObj.getInterpolatedIndex();

          newIdx = virtualPadIndex;
          //newIdx = padIndex;
          bTouchStabilized = false;

          if( bUseProcessing )
            printAllSensors();

          if( DEBUG ) {
            Serial.print("R - "); // R for released
            printAllSensors();
            Serial.print("REAL: ");
            Serial.print( padIndex );
            Serial.print(", VIRTUAL: ");
            Serial.print( virtualPadIndex );
            Serial.println();
          }

          if( b2VVVV ) {
            sentToVVVV( virtualPadIndex );
          }

          if( b2SC && virtualPadIndex!=-1) {
            Serial.print( virtualPadIndex );
            Serial.print('a');
            //Serial.println();
          }
```

```
          if( bUseLEDs ) {
            //carriage.setNewPos( virtualPadIndex );
            carriage.setNewPos( newIdx );
            if( DEBUG ) {
              carriage.debug();
            }
          }
        }
      }
      // reset our state
      mpr[i].lasttouched = mpr[i].currtouched;
    }

    mpr[i].oor = mpr[i].cap.getOOR();

    // SEND DATA TO PROCESSING ****************************************************/
    if( bToPlotter ) {
      sendDataToProcessingPlotter(i);
    }
    //mpr[i].cap.printOOR(); // added for debug purposes
  }

  if( bUseLEDs ) carriage.update();



  // put here a function to control the timing
  // to send send messages to VVVV
  if(newIdx != -1) {

    if(newIdx != oldIdx ) {
      oldIdx = newIdx;
      startWaiting = millis();
    }

    if(millis() - startWaiting > waitToSend && !bTouchStabilized)
    {
      bTouchStabilized = true;
      //Serial.print("touch stabilized - ");
      Serial.write( newIdx );
      //Serial.println();
    }
  }

  delay(DELAY_TIME); // put a delay so it isn't overwhelming
}

/************************************************************************************
 * SERIAL UTILITIES
 ***********************************************************************************/
void getSerialData()
{
  if(Serial.available())
  {
    byte c = Serial.read();
    if (c == 'o')
      bToPlotter = true;
    else if (c == 'c')
      bToPlotter = false;
    else if (c == 'r')
    {
      // reset all the MPR
      for(int i=0; i<NMPR; i++)
        mpr[i].cap.reset();
    }
  }
}

/*
// SERIAL EVENT //////////////////////////////////////////////////////////////////////
// This function cannot be used on Arduino micro
void serialEvent()
{
  byte c = Serial.read();
  if (c == 'o')
    bToPlotter = true;
  else if (c == 'c')
    bToPlotter = false;
  else if (c == 'r')
  {
    // reset all the MPR
    for(int i=0; i<NMPR; i++)
      mpr[i].cap.reset();
  }
}
*/

/************************************************************************************
 * COMPOSE INDEX
 ************************************************************************************/
int composeIndex(int mprIndex, int padIndex) {
  int acc = 0;
  for(int i=0; i<mprIndex; i++) {
    acc += NPADS[i];
  }
  return acc + padIndex;
}


/************************************************************************************
 * PRINT ALL SENSORS
 ************************************************************************************/
void printAllSensors()
{
  // cycle through all the mpr
  for(int i=0; i<NMPR; i++)
  {
    // cycle through all the electrodes
    for(int j=0; j<NPADS[i]; j++)
    {
      int state = (mpr[i].currtouched & _BV(j)) >> j;
      Serial.print( state );
    }
  }
  Serial.println(";");
}

/************************************************************************************
 * SEND TO VVVV
 ************************************************************************************/
void sentToVVVV(int activeVirtualPad) {
  /*
  for( int i=0; i<NVIRTUALPADS; i++) {
    if(i == activeVirtualPad)
      Serial.print(1);
    else
      Serial.print(0);
  }
  Serial.println();
  */
  Serial.println( activeVirtualPad );
```

```
}



// SEND DATA TO PROCESSING PLOTTER //////////////////////////////////////////////////
void sendDataToProcessingPlotter( int mprIndex )
{

  // Send data via serial:
  // 1. First send a byte containing the address of the mpr + the address of the pad +
  //    the 'touched' status of the pad; This byte has a value greater than 127 by convention;
  // 2. Then send two more bytes for 'baseline' and 'filtered' respectively.
  //    Because these values are 10bit values and we must send them
  //    inside a 7bit packet, we must made a 3 times bit shift to the right (/8).
  //    This way we will loose some precision but this is not important.
  //    This two other bytes have values lesser than 127 by convention.

  // cycle all the electrodes
  for(int j=0; j<NPADS[mprIndex]; j++)
  {
    filt = mpr[ mprIndex ].cap.filteredData(j);
    base = mpr[ mprIndex ].cap.baselineData(j);
    b = (1<<7) | (i<<5) | (j<<1) | mpr[ mprIndex ].bPadState[j];
    Serial.write(b); // send address & touched
    Serial.write(base / 8); // send base value
    Serial.write(filt / 8); // send filtered value
  }
}


/*
void sendDataToProcessingPlotter()
{

// ### NEW COMMUNICATION PROTOCOL (19-02-2018) ###
  //
  // Send data via serial:
  // 1. 'Status Byte': first we send a byte containing the address of the mpr.
  //    The most significant bit of the byte is 1 (value of the byte is > 127).
  //    This is a convention in order for the receiver program to be able to recognize it;
  // 2. Then we send 'Data Bytes'. The most significant bit of these bytes is
  //    always 0 in order to differenciate them from the status byte.
  //    We can send as many data bytes as we want. The only thing to keep in mind
  //    is that we must be coherent the receiver side in order not to create confusion
  //    in receiving the data.
  //
  //    For instance we can send pais of byte containing the 'baseline' and 'filtered'
  //    data for each mpr pad.
  //
  //    We can also use data bytes for sending information as:
  //    * 'touched' register;
  //    * 'oor' register;


  // 1. write the status byte containing the mpr addr
  b = (1<<7) | i;
  Serial.write(b);
  // 2. write 'touched' register
  b = mpr[i].currtouched & 0x7F;
  Serial.write(b); //touch status: pad 0 - 6
  b = (mpr[i].currtouched>>7) & 0x7F;
  Serial.write(b); //touch status: pad 7 - 12 (eleprox)
  // 3. write 'oor' register
  b = mpr[i].oor & 0x7F;
  Serial.write(b); //oor status: pad 0 - 6
  b = (mpr[i].oor>>7) & 0x7F;
  Serial.write(b); //oor status: pad 7 - 12 (eleprox)

  // Cycle all the electrodes and send pairs of
  // 'baseline' and 'filtered' data. Mind the bit shifting!
  for(int j=0; j<NPADS; j++)
  {
    base = mpr[i].cap.baselineData(j);
    filt = mpr[i].cap.filteredData(j);
    Serial.write(base>>3); // baseline is a 10 bit value
    Serial.write(filt>>3); // sfiltered is a 10 bit value
  }
}
*/
```

CARRIAGE H

```
/*
*  The "Carriage" class
* Created by Nicola Ariutti, May 17, 2019
* This class contains all the logic to light up LEDS and
* carriage movement (using the Vehicle support class).
* CC0
*/
#ifndef _NA_ABOCA_TIMELINE_CARRIAGE_
#define _NA_ABOCA_TIMELINE_CARRIAGE_
#include "Arduino.h"
#include "Adafruit_NeoPixel.h"
#include "NicolaAriutti_Vehicle.h"
#include "Animator_Sine.h"

class Carriage
{
public:
  Carriage() {};

  void init(Adafruit_NeoPixel *strip, int nVirtPads);
  void update();
  void setNewPos(int _newPos);
  void debug();

private:
  Adafruit_NeoPixel *strip;
  Vehicle *vehicle;
  Animator_Sine *arsine;

  enum {
    STANDBY = 0,
    ACTIVE
  } state = STANDBY;
  long TIMETOWAIT = 5000; // time to wait before moving from ACTIVE to STANDBY
  long prevTime = 0;

  int nLeds;
  int nVirtPads;
  int selectedVirtualPad;

  // the position of the carriage LEDS relative (between 0 and nLeds)
  int centroidPos;
  // must define sidelobe dimension first
  // this is a monolateral dimension
  const int SIDELOBE = 4;
  // then we declare an array that will work as a lookup table
  // for the squared-raised-cosine figure.
  float* lookup = new float[SIDELOBE];

  //int centroidLeds = 0;  // an integer from 0 to NLEDS
  int blue = 0;
};
#endif
```

CARRIAGE CPP

```
#include "Carriage.h"

void Carriage::init(Adafruit_NeoPixel *_strip, int _nVirtPads)
{
  strip = _strip;
  nVirtPads = _nVirtPads;

  // fill the lookup table
  for(int i=0; i<SIDELOBE; i++) {
    int x = i+1;
    lookup[i] = 0.5 * (cos ( (PI*x)/SIDELOBE ) + 1 );
    lookup[i] = lookup[i]*lookup[i];
  }

  nLeds = strip->numPixels();

  // LED STUFF ******************************************************************/
  strip->begin(); // Initialize pins for output

  // set every pixel to sleep
  for(int i=0; i<nLeds; i++) {
    strip->setPixelColor(i, 0x00, 0x00, 0x00);
  }
  strip->show();  // Turn all LEDs off ASAP

  // ANIMATION / VEHICLE ********************************************************/
  // We must calculate this parameter with extreme precision
  // also taking into account the number of leds in the strip
  // and the DELAY_TIME used for each loop cycle.
  vehicle = new Vehicle();
  // Initial position, maxspeed, maxforce, damp distance.
  centroidPos = 0.5*nLeds;
  vehicle->init( centroidPos, 0.5, 0.1, 1.5);
  //int initialPos = (((7 * 1.0)/nVirtPads) * (nLeds) );

  arsine = new Animator_Sine();
  arsine->init(1, 0.0);
}

void Carriage::update()
{
  if( state == STANDBY )
  {
    arsine->update();
    float y = arsine->getY();
    y = y * y;
    byte color;
    for(int i=0; i<nLeds; i++) {
      int distance = abs(i-centroidPos);
      if(distance == 0) {
        color = y * 255.0;
      }
      else if( distance <= SIDELOBE ) {
        color = lookup[distance-1]*255.0 *y;
      }
      else {
        blue = 0;
      }
      strip->setPixelColor(i, color, color, color);
    }
    //strip->show();
    strip->show();
  }
  else if(state == ACTIVE && (millis() - prevTime) > TIMETOWAIT) {
    state = STANDBY;
    prevTime = millis();
  }
  else
  {
    // VEHICLE STUFF
    // calculate the scaled centroid

    vehicle->update();
    centroidPos = vehicle->getPosition();
    //Serial.println(pos);

    // LED STUFF
    // The logic below is created in order to properly light-up
    // leds around the centroid according to the
    // squared raised-cosine lookup table.
    for(int i=0; i<nLeds; i++) {
      int distance = abs(i-centroidPos);
      if(distance == 0) {
        blue = 255;
      }
      else if( distance <= SIDELOBE ) {
        blue = lookup[distance-1]*255.0;
      }
      else {
        blue = 0;
      }
      strip->setPixelColor(i, blue, blue, blue);
    }
    strip->show();
  }
}

void Carriage::setNewPos(int _selectedVirtualPad )
{
  // time
  state = ACTIVE;
  prevTime = millis();

  // if the touch object is not EMPTY
  if( _selectedVirtualPad != -1) {
    selectedVirtualPad = _selectedVirtualPad;
    centroidPos = (((1.0 *  selectedVirtualPad)/nVirtPads) * (nLeds - 2*SIDELOBE +1))+SIDELOBE;
    vehicle->setTarget( centroidPos );
  }
}

void Carriage::debug() {
  Serial.print("STATUS: ");
  if(state == 0)
    Serial.print("STANDBY");
  else if(state == 1)
    Serial.print("ACTIVE");
  Serial.print(", centroids: PAD ");
  Serial.print(selectedVirtualPad);
  Serial.print(", LEDs ");
  Serial.print(centroidPos);
  Serial.println();
}
```

VEHICLE H

```
/*
*  The "Vehicle" class
* Created by Nicola Ariutti, March 06, 2018
* CC0
*
* Inspired by
* The Nature of Code
* Daniel Shiffman
* http://natureofcode.com
*/
#ifndef _NICOLAARIUTTI_VEHICLE
#define _NICOLAARIUTTI_VEHICLE
#include "Arduino.h"

class Vehicle
{
public:
        Vehicle() {};

  // Initial position, maxspeed, maxforce, damp distance.
        void init( float _x, float _ms, float _mf, int _damp );
  void update();
        void setTarget(float _target);
  float getPosition();

private:
        float position;
  float velocity;
  float acceleration;

        float maxforce;    // Maximum steering force
  float maxspeed;    // Maximum speed

  float target;
  int DAMPING_DISTANCE;
};
#endif
```

VEHICLE CPP

```
#include "NicolaAriutti_Vehicle.h"

void Vehicle::init(float _x, float _ms, float _mf, int _damp) {
        acceleration = 0.0;
        velocity = 0.0;
        position = _x;
        target = position;

        maxspeed = _ms;
        maxforce = _mf;
        DAMPING_DISTANCE = _damp;
}

// Method to update position
void Vehicle::update() {
        float desired = target - position;  // A float pointing from the position to the target
        float d = abs(desired);

        if( d == 0.0 )
        {
                // we are already there
                position = target;
                return;
        }
        // Scale with arbitrary damping within 100 pixels
        else if (d < DAMPING_DISTANCE)
        {
                float m = map(d,0,DAMPING_DISTANCE,0,maxspeed);
                desired = (desired * m)/d;
        }
        else
        {
                desired = (desired * maxspeed)/d;
        }

        // Steering = Desired minus Velocity
        float steer = desired - velocity;
        if(abs(steer) >= maxforce)
                steer = (steer*maxforce)/abs(steer); // Limit to maximum steering force

        // We could add mass here if we want A = F / M
        acceleration += steer;

        // Update velocity
        velocity +=acceleration;
        // Limit speed
        if(abs(velocity) >= maxspeed)
                velocity = (velocity*maxspeed)/abs(velocity);
        // Update position
        position += velocity;
        // Reset accelerationelertion to 0 each cycle
        acceleration = 0.0;
}


// A method that calculates a steering force towards a target
// STEER = DESIRED MINUS VELOCITY
void Vehicle::setTarget(float _target) {
        target = _target;
}

float Vehicle::getPosition() {
        return position;
}
```

TOUCH H

```
/*
*  The "Touch" class
* Created by Nicola Ariutti, May 17, 2019
* CC0
*/
#ifndef _NA_TOUCH_
#define _NA_TOUCH_
#include "Arduino.h"

class Touch
{
public:
  Touch() {};

  void addIndex(int _index);
  void removeIndex(int _index);
  int getInterpolatedIndex();

private:
  enum {
    EMPTY = 0,
    SINGLETOUCH,
    DOUBLETOUCH,
    TRIPLETOUCH
  } touchStatus = EMPTY;
  int idxL = -1;
  int idxM = -1;
  int idxR = -1;
};
#endif
```

TOUCH CPP

```
#include "Touch.h"

void Touch::addIndex(int _index) {
  if( touchStatus == EMPTY) {
    idxL = _index;
    touchStatus = SINGLETOUCH;
  }
  else if( touchStatus == SINGLETOUCH  ) {
    // the new index is adjacent to the one already touched
    if( _index == idxL-1 ) {
      idxR = idxL;
      idxL = _index;
      touchStatus = DOUBLETOUCH;
    }
    else if (_index == idxL+1) {
      idxR = _index;
      touchStatus = DOUBLETOUCH;
    }
    else
    {
      // the new index is not adjacent
      idxL = _index;
      // status remains the same
      touchStatus = SINGLETOUCH; // redundant
    }
  } else if( touchStatus == DOUBLETOUCH) {
    // if we are here it means we have a third consecutive touch without a release.

    // check if the incoming index is to the left of minimumum between 1st and 2nd
    // indexes or if it is to the right of maximum between the two
    if( _index == idxL-1 ) {
      idxM = idxL;
      idxL = _index;
      touchStatus = TRIPLETOUCH;
    } else if ( _index == idxR+1) {
      idxM = idxR;
      idxR = _index;
      touchStatus = TRIPLETOUCH;
    } else {
      // if we are here it means that the third touch is elsewhere
      // so we will treat it as a brand new single touch
      idxL = _index;
      idxR = -1;
      touchStatus = SINGLETOUCH;
    }
  } else if( touchStatus == TRIPLETOUCH ) {
      if( _index == idxL-1) {
        idxR = idxM;
        idxM = idxL;
        idxL = _index;
        touchStatus = TRIPLETOUCH; // redundant
      }
      else if (_index == idxR+1) {
        // if so, exchange indexes (discard the oldest index, always keep only two indexes)
        idxL = idxM;
        idxM = idxR;
        idxR = _index;
        touchStatus = TRIPLETOUCH; // redundant
      }
      else
      {
        // if we are here it means that the third touch is elsewhere
        // so we will treat it as a brand new single touch
        idxL = _index;
        idxR = -1;
        idxM = -1;
        touchStatus = SINGLETOUCH;
      }
    }
}


void Touch::removeIndex(int _index) {
  // it can happen this method to be called even if
  // the index to be removed isn't contained inside
  // the touch object because it only keeps track of single
  // touches or a couple of adjacent ones.
  // So do nothing in that cases.
  if( touchStatus == SINGLETOUCH && _index == idxL)
  {
    idxL = -1;
    touchStatus = EMPTY;
  }
  else if ( touchStatus == DOUBLETOUCH )
  {
    if(_index == idxL) {
      idxL = idxR;
      idxR = -1;
      touchStatus = SINGLETOUCH;
    }
    else if (_index == idxR ) {
      idxR = -1;
      touchStatus = SINGLETOUCH;
    }
  }
  else if( touchStatus == TRIPLETOUCH ) {
    if (_index == idxL) {
      idxL = idxM;
      idxM = -1;
      touchStatus = DOUBLETOUCH;
    } else if (_index == idxR) {
      idxR = idxM;
      idxM = -1;
      touchStatus = DOUBLETOUCH;
    } else if (_index == idxM) {
      // in a case like this one i deliberately
      // choose to maintain only the left index
      idxM = -1;
      idxR = -1;
      touchStatus = SINGLETOUCH;
    }
  }
}


// a new interpolation function in order to use not 48 index
// but only 25.
int Touch::getInterpolatedIndex() {
  if( touchStatus == SINGLETOUCH ) {
    return idxL;
  }
  else if( touchStatus == DOUBLETOUCH ) {
    return int((idxL+idxR)*0.5);
  }
  else if( touchStatus == TRIPLETOUCH ) {
    return idxM; // same as doing idxL + idxR;
  }
  /*
  else if( touchStatus == EMPTY ){
    //Serial.println("EMPTY");
    // this should never happen
    return -1;
  }
  */
  return -1;
}



/*
int Touch::getInterpolatedIndex() {
  if( touchStatus == SINGLETOUCH ) {
    return idxL * 2;
  }
  else if( touchStatus == DOUBLETOUCH ) {
    return idxL+idxR;
  }
  else if( touchStatus == TRIPLETOUCH ) {
    return idxM * 2; // same as doing idxL + idxR;
  }

  //else if( touchStatus == EMPTY ){
  //  //Serial.println("EMPTY");
  //  // this should never happen
  //  return -1;
  //}

  return -1;
}
*/
```