```
#include "gstvideo.h"

guintptr gstvideo::cam_window_handle;
static int effect = 0;
static GstPad  *binpad;
static GstPad *blockpad;



gstvideo::gstvideo(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::gstvideo)
{

    ui->setupUi(this);
    ui->slider1->setRange(-100,100);//contrast 0 -> 2. default=1
    this->setWindowTitle("BitRiver");
    ui->slider1->setTickPosition(QSlider::TicksAbove);
    ui->slider2->setRange(-100,100);//brightness    -1 -> 1. default=0
    ui->slider2->setTickPosition(QSlider::TicksAbove);
    ui->slider3->setRange(-100,100);//saturation 0 -> 2. default=1
    ui->slider3->setTickPosition(QSlider::TicksAbove);
    ui->slider4->setRange(-100,100);//hue       -1 -> 1. default=0
    ui->slider4->setTickPosition(QSlider::TicksAbove);
    ui->slider1->setValue(0);
    ui->slider2->setValue(0);
    ui->slider3->setValue(0);
    ui->slider4->setValue(0);
    ui->slider5->setRange(0,10);
    ui->slider5->setValue(0);
    ui->slider5->setTickPosition(QSlider::TicksAbove);
    ui->progressBar1->setValue(0);
    ui->progressBar1->setRange(-100,100);
    ui->progressBar2->setValue(0);
    ui->progressBar2->setRange(-100,100);
    ui->progressBar3->setValue(0);
    ui->progressBar3->setRange(-100,100);
    ui->progressBar4->setValue(0);
    ui->progressBar4->setRange(-100,100);
    ui->comboBox->addItems(QStringList()<<"identity" <<"dicetv"
                           <<"warptv"<<"shagadelictv"<< "revtv"<< "radioactv"<<
                           "rippletv"<<"TehRoxx0r"<<"Cartoon"<<"invert"
                          <<"pixeliz0r"<<"Nervous"<<"Vertigo"<<"Color Distance"<<"perspective"<<"color-
                          B"<<"Baltan"<<"Twolay0r"<<"threelay0r"
                          <<"bw0r"<<"Sobel"<<"Distort0r");

    QObject::connect(ui->slider1, SIGNAL(valueChanged(int)),
                     ui->progressBar1, SLOT(setValue(int)));
    QObject::connect(ui->slider2, SIGNAL(valueChanged(int)),
                     ui->progressBar2, SLOT(setValue(int)));
    QObject::connect(ui->slider3, SIGNAL(valueChanged(int)),
                     ui->progressBar3, SLOT(setValue(int)));
    QObject::connect(ui->slider4, SIGNAL(valueChanged(int)),
                     ui->progressBar4, SLOT(setValue(int)));
    ui->widget->setFixedWidth(640);
    ui->widget->setFixedHeight(480);
    gst_init(NULL, FALSE);
    input->exec();
    g_print("video Resolution: %dx%d \n", input->resolutionX, input->resolutionY);
    g_print("audio rate is: %d; audio bitrate is: %d \n", input->arate, input->abrate);
    g_print("video settings - framerate: %d, video bitrate: %d \n",input->framerate, input->vbrate);
    g_print("audio channels is: %d \n", input->channels);

    this->conversor1 = gst_element_factory_make("videoconvert", "conversor1");
    this->conv = gst_element_factory_make("audioconvert","aconv");
    this->volume = gst_element_factory_make("volume","volume");
    this->x264enc = gst_element_factory_make("x264enc","x264enc");
    this->h264parse = gst_element_factory_make("h264parse","h264parse");
    this->avdec_h264 = gst_element_factory_make("avdec_h264","avdec_h264");
    this->sink = gst_element_factory_make("ximagesink", "sink");
    this->videobalance = gst_element_factory_make("videobalance", "balance");// ########################
    this->audiosink = gst_element_factory_make("autoaudiosink", "ausink");
    this->faac = gst_element_factory_make("voaacenc","aacAudioencoder");
    this->aacparse = gst_element_factory_make("aacparse", "aacparse");

    this->audiosampler = gst_element_factory_make("audioresample", "audiosampler");
    this->curr = gst_element_factory_make("identity", "curr");
    this->conv_after = gst_element_factory_make("videoconvert", "conv_after");
    this->conv_before = gst_element_factory_make("videoconvert", "conv_before");
    this->queue1 = gst_element_factory_make("queue", "queue1");
    this->queue2 = gst_element_factory_make("queue", "queue2");
    this->queue3 = gst_element_factory_make("queue", "queue3");
    this->queue4 = gst_element_factory_make("queue", "queue4");
    this->queue5 = gst_element_factory_make("queue", "queue5");
    this->queue6 = gst_element_factory_make("queue", "queue6");
    this->queue7 = gst_element_factory_make("queue", "queue7");
    this->queue8 = gst_element_factory_make("queue", "queue8");
```

```
    this->queue9 = gst_element_factory_make("queue", "queue9");
    this->Ltee2 = gst_element_factory_make("tee","tee2");//audio branch tee for visualization
    this->Ltee1 = gst_element_factory_make("tee","tee1");//video branch tee for visualization
    this->scale = gst_element_factory_make("videoscale","scale");//for video streaming settings
    this->videosinkconvert = gst_element_factory_make("videoconvert", "vsinkconvert");
    this->Svideoconvert = gst_element_factory_make("videoconvert", "sconvert");
    this->videorate = gst_element_factory_make("videorate", "videorate");
    this->audiorate = gst_element_factory_make("audiorate", "audiorate");
    //this->audiosinkconvert = gst_element_factory_make("audioconvert","audiosinkconvert");
    this->audioparse = gst_element_factory_make("audioparse", "audiopar");
    this->pipeline = gst_pipeline_new("pipeline");
    this->rtmp = gst_element_factory_make("fakesink","rtmp");
    this->flvmux = gst_element_factory_make("flvmux","flvmux");

    int keyint = 2*input->framerate;
    QString location = "rtmp://a.rtmp.youtube.com/live2/x/" + input->youtube + "?
    videoKeyframeFrequency=1&totalDatarate=8128 app=live2 flashVer=FME/3.0%20(compatible;%20FMSc%201.0)
    swfUrl=rtmp://a.rtmp.youtube.com/live2";
    g_object_set(this->rtmp, "location", location.toUtf8().constData(), "sync", FALSE, NULL);

    g_object_set(this->volume, "volume", 0, NULL);
    g_object_set(this->faac, "bitrate", input->abrate, NULL);
    g_object_set(this->x264enc, "bitrate", input->vbrate, "key-int-max", keyint, "bframes", 0, "byte-
    stream", false, "aud", true,
                 "threads", 2, "speed-preset", 1, "pass", 17, NULL);
    g_object_set(this->sink, "sync", TRUE, NULL);
    g_object_set(this->audiosink, "sync", TRUE, NULL);
    g_object_set(this->audioparse, "rate", input->arate, "channels", input->channels, NULL);
    g_object_set(this->flvmux, "streamable", TRUE, NULL);

    this->Vcaps = gst_caps_new_simple("video/x-raw",
                                      //"format", G_TYPE_STRING, "BGRA",
                                      //"framerate", GST_TYPE_FRACTION, 25, 1,
                                      "interlace-mode", G_TYPE_STRING, "progressive",
                                      "width", G_TYPE_INT, 640,
                                      "height", G_TYPE_INT, 480,
                                       NULL);
    this->Scaps = gst_caps_new_simple("video/x-raw",
                                      //"framerate", GST_TYPE_FRACTION, 25, 1,
                                      "interlace-mode", G_TYPE_STRING, "progressive",
                                      "width", G_TYPE_INT, input->resolutionX,
                                      "height", G_TYPE_INT, input->resolutionY,
                                       NULL);
    this->Acaps = gst_caps_new_simple("audio/x-raw",
                                      "format", G_TYPE_STRING, "S16LE",
                                      "layout", G_TYPE_STRING, "interleaved",
                   NULL);

    this->enVcaps = gst_caps_new_simple("video/x-h264",
                   "level", G_TYPE_STRING, "4.1",
                   "profile", G_TYPE_STRING, "main",
                   NULL);

    this->enAcaps = gst_caps_new_simple("audio/mpeg",
                   "mpegversion", G_TYPE_INT, 4,
                   "stream-format", G_TYPE_STRING, "raw",
                    NULL);

    // ########### Checking for errors
    ################################################################################

    if (!pipeline ){
        qDebug("pipeline not created");
        return;
    }
    if (!this->conv || !this->volume || !this->audiosampler || !this->faac || !this->aacparse || !this-
    >audiosink
            || !this->audiorate || !this->audioparse){
        qDebug("any audio element not found");
        return;
    }


    if (!this->x264enc || !this->h264parse || !flvmux || !Ltee1 || !Ltee2 || !queue1 || !queue2 || !queue3
    || !queue4
            || !queue5 || !queue6 || !queue7 || !queue8 || !queue9 || !this->scale || !this->enAcaps || !
            this->videobalance
             || !this->conversor1 || !conv_before || !curr || !conv_after  || !rtmp || !Vcaps || !Scaps){
        qDebug("any video or encoding element not found");
        return;
    }


    //
    ##############################################################################################################
    ##############################
```

```
    binpad = gst_element_get_static_pad(this->volume, "src");
    GstPad *pad = gst_element_get_static_pad(this->videobalance, "src");
    //GstPad *pada = gst_element_get_static_pad(this->conv, "audioconvert");

            switch (input->videoBIN){

            case 1://tcp input source

            {
                this->vdecoder = gst_element_factory_make("decodebin","vdecodebin");
                this->Vtcpsrc = gst_element_factory_make("tcpclientsrc", "Vtcpsrc");
                g_object_set(this->Vtcpsrc, "host", input->videotcp.toUtf8().constData(), "port", input->vport,NULL);

                blockpad = gst_element_get_static_pad(queue1, "src");

                this->Vscale = gst_element_factory_make("videoscale","Vscale");
                this->Sscale = gst_element_factory_make("videoscale","Sscale");

                gst_bin_add_many(GST_BIN(pipeline), this->Vtcpsrc, vdecoder, queue1, this->scale, this->conversor1,
                                 this->videobalance, conv_before, curr, conv_after,this->Ltee1, queue7,
                                 this->Sscale,
                                 this->Svideoconvert,this->x264enc,
                                 this->h264parse, queue3, this->flvmux, queue4,
                                 this->rtmp, queue2,this->conv, this->audiosampler, this->volume, this->Ltee2,
                                 queue9, this->faac, this->aacparse, queue5,
                                 queue8, this->audiosink,
                                 queue6, this->Vscale, this->videosinkconvert, this->sink, NULL);

                gst_element_link_many(this->Vtcpsrc, vdecoder, NULL);
                gst_element_link_many(queue1, this->scale, this->conversor1,NULL);
                gst_element_link_filtered (this->conversor1, this->videobalance ,this->Scaps);
                gst_element_link_many(this->videobalance,
                                      conv_before, curr, conv_after,this->Ltee1, NULL);
                gst_element_link_many(queue6, this->Vscale, this->videosinkconvert, NULL);
                gst_element_link_filtered(this->videosinkconvert, this->sink, this->Vcaps);
                gst_element_link_many(queue7, this->Sscale, this->Svideoconvert, NULL);
                gst_element_link(this->Svideoconvert, this->x264enc);
                gst_element_link_many(this->x264enc, this->h264parse, queue3, NULL);
                gst_element_link_many(this->flvmux, queue4, this->rtmp, NULL);
                gst_element_link_many(queue2, this->conv, this->audiosampler, this->volume, Ltee2, NULL);
                gst_element_link_many(queue8, this->audiosink, NULL);
                gst_element_link_many(queue9, this->faac, this->aacparse, NULL);
                gst_element_link_filtered(this->aacparse, queue5, this->enAcaps);

                GstPadTemplate *tee_src_pad_template1, *tee_src_pad_template2;
                GstPad *tee1_q6_pad, *tee1_q7_pad,*tee2_q8_pad, *tee2_q9_pad;
                  GstPad *q6_pad, *q7_pad, *q8_pad, *q9_pad;


                if ( !(tee_src_pad_template1 = gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS(this->Ltee1), "src_%u"))) {
                 gst_object_unref (pipeline);
                 g_critical ("Unable to get pad template");
                }

                if ( !(tee_src_pad_template2 = gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS(this->Ltee2), "src_%u"))) {
                 gst_object_unref (pipeline);
                 g_critical ("Unable to get pad template");
                }

                tee1_q7_pad = gst_element_request_pad (this->Ltee1, tee_src_pad_template1, NULL, NULL);

                 q7_pad = gst_element_get_static_pad (queue7, "sink");
                 /* Link the tee to the queue 7 */
                 if (gst_pad_link (tee1_q7_pad, q7_pad) != GST_PAD_LINK_OK ){ // t2 ----> queue7
                  g_critical ("Tee1 for queue7 could not be linked.\n");
                  gst_object_unref (pipeline);
                  exit(1);
                 }


                 tee2_q9_pad = gst_element_request_pad (this->Ltee2, tee_src_pad_template2, NULL, NULL);

                  q9_pad = gst_element_get_static_pad (queue9, "sink");
                  /* Link the tee to the queue 7 */
                  if (gst_pad_link (tee2_q9_pad, q9_pad) != GST_PAD_LINK_OK ){
                   g_critical ("Tee2 for queue9 could not be linked.\n");
                   gst_object_unref (pipeline);
                   exit(1);
                  }
```

```
                tee1_q6_pad = gst_element_request_pad (this->Ltee1, tee_src_pad_template1, NULL, NULL);

                q6_pad = gst_element_get_static_pad(queue6, "sink");
                /* Link the tee to the queue 6 */
                if (gst_pad_link(tee1_q6_pad, q6_pad) != GST_PAD_LINK_OK ){ //  t1 ----> queue6
                 g_critical ("Tee1 for queue6 could not be linked.\n");
                 gst_object_unref (pipeline);
                 exit(1);
                }

                /* Obtaining request pads for the tee1 elements*/
                tee2_q8_pad = gst_element_request_pad (this->Ltee2, tee_src_pad_template2, NULL, NULL);
                q8_pad = gst_element_get_static_pad (queue8, "sink");
                /* Link the tee to the queue 6 */
                if (gst_pad_link (tee2_q8_pad, q8_pad) != GST_PAD_LINK_OK ){
                 g_critical ("Tee2 for queue8 could not be linked.\n");
                 gst_object_unref (pipeline);
                 exit(1);
                 }



                GstPadTemplate *flvmux_sink_pad_template_audio;
                if (!(flvmux_sink_pad_template_audio =
                gst_element_class_get_pad_template(GST_ELEMENT_GET_CLASS(this->flvmux), "audio"))) {
                    gst_object_unref (pipeline);
                    printf ("Unable to get pad template for audio for flvmux element");
                    exit(1);
                }

                GstPad * audio_queue5_src_pad = gst_element_get_static_pad(queue5, "src");
                GstPad * flvmux_sink_audio_pad = gst_element_request_pad(flvmux,
                flvmux_sink_pad_template_audio, NULL, NULL);
                if (gst_pad_link (audio_queue5_src_pad, flvmux_sink_audio_pad) != GST_PAD_LINK_OK ) {
                    printf("unable to link audio queue with flvmixer\n");
                    exit(1);
                }

               GstPadTemplate *flvmux_sink_pad_template_video;
                   if (!(flvmux_sink_pad_template_video =
                   gst_element_class_get_pad_template(GST_ELEMENT_GET_CLASS(this->flvmux), "video"))) {
                       gst_object_unref (pipeline);
                       printf ("Unable to get pad template for video for flvmux element");
                       exit(1);
                   }

                   GstPad * video_queue3_src_pad = gst_element_get_static_pad(queue3, "src");
                   GstPad * flvmux_sink_video_pad = gst_element_request_pad(this->flvmux,
                   flvmux_sink_pad_template_video, NULL, NULL);
                   if (gst_pad_link(video_queue3_src_pad, flvmux_sink_video_pad) == GST_PAD_LINK_OK ) {
                       printf("link video queue with flvmixer\n");
                   }

                   gst_object_unref(audio_queue5_src_pad);
                   gst_object_unref(video_queue3_src_pad);
                   gst_object_unref (q6_pad);
                   gst_object_unref(q7_pad);
                   gst_object_unref (q8_pad);
                   gst_object_unref(q9_pad);
                   gst_object_unref(binpad);
                   gst_object_unref(pad);
                   gst_object_unref(tee_src_pad_template1);
                   gst_object_unref(tee_src_pad_template2);
        }


            break;
         default:
            break;
        }


    //end of else for input->isLocal evaluation

    window = ui->widget->winId();

    cam_window_handle=window;
    this->bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
    gst_bus_set_sync_handler (this->bus, (GstBusSyncHandler) bus_sync_handler, this, NULL);
    gst_object_unref (this->bus);

    GstBus *mbus = gst_pipeline_get_bus(GST_PIPELINE(this->pipeline));
    gst_bus_add_signal_watch(mbus);

    connect(ui->slider1, SIGNAL(valueChanged(int)), this, SLOT(contrast(int)));
    connect(ui->slider2, SIGNAL(valueChanged(int)), this, SLOT(brightness(int)));
```

```
    connect(ui->slider3, SIGNAL(valueChanged(int)), this, SLOT(hue(int)));
    connect(ui->slider4, SIGNAL(valueChanged(int)), this, SLOT(saturation(int)));
    connect(ui->slider5, SIGNAL(valueChanged(int)), this, SLOT(avolume(int)));
    connect(ui->bplay, SIGNAL(clicked()), this, SLOT (start()));
    connect(ui->bstop, SIGNAL(clicked()), this, SLOT(stop()));
    g_signal_connect(vdecoder, "pad-added", G_CALLBACK(videoPad_added_handler), this);//### REVISAR AQUI
PARA VER EL vdecoder
    g_signal_connect(mbus, "message::error", G_CALLBACK(callback), this);

    delete input;
}


gstvideo::~gstvideo()
{
    gst_element_set_state(GST_ELEMENT(pipeline), GST_STATE_NULL);
    gst_object_unref(pipeline);
    delete ui;
}

void gstvideo::callback(GstBus *bus, GstMessage* msg, gstvideo* v)
{
    g_print("Got %s message\n", GST_MESSAGE_TYPE_NAME(msg));
    GError *err;
    gchar *debug;
    gst_message_parse_error(msg, &err, &debug);
    g_print("from %s \n", GST_MESSAGE_SRC_NAME(msg));
    g_print("Error: %s\n", err->message);
    g_error_free(err);
    g_free(debug);
}

//sync callback function for ximagesink and qt widget
GstBusSyncReply gstvideo::bus_sync_handler (GstBus *bus, GstMessage *message, gstvideo *v)
{
    if (!gst_is_video_overlay_prepare_window_handle_message (message))
        return GST_BUS_PASS;

    GstVideoOverlay *overlay;
    overlay = GST_VIDEO_OVERLAY (GST_MESSAGE_SRC (message));
    gst_video_overlay_set_window_handle (overlay, cam_window_handle);
    gst_message_unref (message);
    return GST_BUS_DROP;
    //arriba añado los elementos creados en el constructor a la pipeline de gstreamer
    //luego los conecto
    //obtengo el ID de la ventana creada en qt la cual asignare al ximagesink de gstreamer
    //y aseguro de sincronizar el llamado de la ventana a traves de mensajes en el bus de gstreamer
}




void gstvideo::update_color_channel (gchar *channel_name, gint dvalue, GstColorBalance *cb) {
  GstColorBalanceChannel *channel = NULL;
  const GList *channels, *l;

  /* Retrieve the list of channels and locate the requested one */
  channels = gst_color_balance_list_channels (cb);
  for (l = channels; l != NULL; l = l->next) {
    GstColorBalanceChannel *tmp = (GstColorBalanceChannel *)l->data;

    if (g_strrstr (tmp->label, channel_name)) {
      channel = tmp;
      break;
    }
  }
  if (!channel)return;

    if (dvalue > channel->max_value)
    {
      dvalue = channel->max_value;
      g_print("%d/n", dvalue);
    }
    else {
        if (dvalue < channel->min_value)
            dvalue = channel->min_value;
         }
   gst_color_balance_set_value(cb, channel, dvalue);

}



GstPadProbeReturn gstvideo::block_src(GstPad *pad, GstPadProbeInfo *info, gstvideo *v){
```

```
    GstPad *srcpad, *sinkpad;
    gst_pad_remove_probe(blockpad, GST_PAD_PROBE_INFO_ID(info));
    /* install new probe for EOS */
    srcpad = gst_element_get_static_pad (v->curr, "src");
    if(!pad)
    {
        g_print("no se pudo obtener el pad del elemento para enviar un EOS");
        exit(1);
     }
    gst_pad_add_probe(srcpad, GST_PAD_PROBE_TYPE_EVENT_DOWNSTREAM, (GstPadProbeCallback)event_eos, v,
    NULL);
    gst_object_unref (srcpad);

    /* push EOS into the element, the probe will be fired when the
     * EOS leaves the effect and it has thus drained all of its data */

    sinkpad = gst_element_get_static_pad (v->curr, "sink");
    gst_pad_send_event (sinkpad, gst_event_new_eos ());
    gst_object_unref (sinkpad);

    return GST_PAD_PROBE_OK;
}


GstPadProbeReturn gstvideo::event_eos(GstPad * pad, GstPadProbeInfo * info, gstvideo *v)
{

  if (GST_EVENT_TYPE (GST_PAD_PROBE_INFO_DATA (info)) != GST_EVENT_EOS)
    return GST_PAD_PROBE_OK;
  gst_pad_remove_probe (pad, GST_PAD_PROBE_INFO_ID (info));
  gst_element_set_state (v->curr, GST_STATE_NULL);
  gst_bin_remove (GST_BIN (v->pipeline), v->curr);

  switch (effect){
  case 0:
      v->curr = gst_element_factory_make("identity", "next");
      break;
  case 1:
      v->curr = gst_element_factory_make("dicetv", "next");
      break;
  case 2:
      v->curr = gst_element_factory_make("warptv", "next");
      break;
  case 3:
      v->curr = gst_element_factory_make("shagadelictv", "next");
      break;
  case 4:
      v->curr = gst_element_factory_make("revtv", "next");
      break;
  case 5:
      v->curr = gst_element_factory_make("radioactv", "next");
      break;
  case 6:
      v->curr = gst_element_factory_make("rippletv", "next");
      break;
  case 7:
      v->curr = gst_element_factory_make("frei0r-filter-tehroxx0r", "next");
      break;
  case 8:
      v->curr = gst_element_factory_make("frei0r-filter-cartoon", "next");
      break;
  case 9:
      v->curr = gst_element_factory_make("frei0r-filter-invert0r", "next");
      break;
  case 10:
      v->curr = gst_element_factory_make("frei0r-filter-pixeliz0r", "next");
      break;
  case 11:
      v->curr = gst_element_factory_make("frei0r-filter-nervous", "next");
      break;
  case 12:
      v->curr = gst_element_factory_make("frei0r-filter-vertigo", "next");
      break;
  case 13:
      v->curr = gst_element_factory_make("frei0r-filter-color-distance", "next");
      break;
  case 14:
      v->curr = gst_element_factory_make("frei0r-filter-perspective", "next");
      g_object_set(v->curr, "top-left-x", 0.8, "top-left-Y", 0.01, "top-right-x",0.01, "top-right-Y",
      0.03490 ,  NULL);
      break;
  case 15:
      v->curr = gst_element_factory_make("frei0r-filter-b", "next");
      break;
  case 16:
      v->curr = gst_element_factory_make("frei0r-filter-baltan", "next");
      break;
```

```
  case 17:
      v->curr = gst_element_factory_make("frei0r-filter-twolay0r", "next");
      break;
  case 18:
      v->curr = gst_element_factory_make("frei0r-filter-threelay0r", "next");
      break;
  case 19:
      v->curr = gst_element_factory_make("frei0r-filter-bw0r", "next");
      break;
  case 20:
      v->curr = gst_element_factory_make("frei0r-filter-sobel", "next");
      break;
  case 21:
      v->curr = gst_element_factory_make("frei0r-filter-distort0r", "next");
      break;
  default:
      v->curr = gst_element_factory_make("identity", "next");
      break;
    }

  gst_bin_add (GST_BIN (v->pipeline), v->curr);
  gst_element_link_many (v->conv_before, v->curr, v->conv_after, NULL);
  gst_element_set_state (v->curr, GST_STATE_PLAYING);
  return GST_PAD_PROBE_DROP;
}

//################################## DINAMIC CALLBACK for vdecoder
############################################################

/* This function will be called by the pad-added signal */

void gstvideo::videoPad_added_handler(GstElement *src, GstPad *new_pad, gstvideo *v) {
    Q_UNUSED(src);
    g_print("entering into padd-added video function: ");
    GstPad *sinkpad = NULL;
    GstPadLinkReturn ret;
    GstCaps *new_pad_caps = NULL;
    GstStructure *new_pad_struct = NULL;
    new_pad_caps = gst_pad_get_current_caps(new_pad);
    new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);
    int pad_count = 0;

    if (g_strrstr (gst_structure_get_name (new_pad_struct), "video")) //checking if there is video caps
       {
        pad_count++;
         sinkpad = gst_element_get_static_pad(v->queue1, "sink");
       }

    else
    {
         sinkpad = gst_element_get_static_pad (v->queue2, "sink"); //it is a audio caps structure
         pad_count++;
     }
    gst_caps_unref (new_pad_caps);

    gst_pad_link (new_pad, sinkpad);
    gst_object_unref (sinkpad);
}

// ##################################### SLOTS
#############################################################################

void gstvideo::start()
{
    gst_element_set_state (this->pipeline, GST_STATE_PLAYING);
    qDebug()<<"the Pipeline State is changing to playing STATE";
}

void gstvideo::stop()
{
    if (pipeline != NULL)
    {
        gst_element_set_state(this->pipeline, GST_STATE_NULL);
        qDebug()<<"the Pipeline State is changing to Paused";
;
    }
}

void gstvideo::contrast(int c){
    c = c*10;
    this->update_color_channel("CONTRAST", c, GST_COLOR_BALANCE(this->videobalance));
}

void gstvideo::brightness(int b){
    b = b*10;
    this->update_color_channel("BRIGHTNESS", b, GST_COLOR_BALANCE(this->videobalance));
}
```

```
void gstvideo::hue(int h){
    h = h*10;
    this->update_color_channel("HUE", h, GST_COLOR_BALANCE(this->videobalance));
}

void gstvideo::saturation(int s){
    s = s*10;
    this->update_color_channel("SATURATION", s, GST_COLOR_BALANCE(this->videobalance));
}


void gstvideo::on_comboBox_currentIndexChanged(int index)
{
     Q_UNUSED(index);
    effect=ui->comboBox->currentIndex();
    //gstvideo *v;
    gst_pad_add_probe(blockpad, GST_PAD_PROBE_TYPE_BLOCK_DOWNSTREAM, (GstPadProbeCallback)block_src,this,
    NULL);
    //GstPadProbeReturn gstvideo::event_eos(GstPad * pad, GstPadProbeInfo * info, gpointer user_data)
}

void gstvideo::avolume(int y){
    gdouble x = y/10.0;
    g_print("%d \n", x);
    gst_stream_volume_set_volume (GST_STREAM_VOLUME(this->volume), GST_STREAM_VOLUME_FORMAT_LINEAR, x);
}
```