```
/*
 * VDPAU video output driver
 *
 * Copyright (C) 2008 NVIDIA
 *
 * This file is part of MPlayer.
 *
 * MPlayer is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * MPlayer is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with MPlayer; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

/**
 * \defgroup  VDPAU_Presentation VDPAU Presentation
 * \ingroup Decoder
 *
 * Actual decoding and presentation are implemented here.
 * All necessary frame information is collected through
 * the "vdpau_render_state" structure after parsing all headers
 * etc. in libavcodec for different codecs.
 *
 * @{
 */

#include <stdio.h>

#include "config.h"
#include "mp_msg.h"
#include "video_out.h"
#include "video_out_internal.h"
#include "x11_stereo.h"
#include "aspect.h"
#include "sub.h"
#include "subopt-helper.h"

#include "libavcodec/vdpau.h"

#include "gui/interface.h"

#include "libavutil/common.h"
#include "libavutil/mathematics.h"

#include "libass/ass.h"
#include "libass/ass_mp.h"

static vo_info_t info = {
    "VDPAU with X11 stereo",
    "vdpaustereo",
    "Cristian Rocha <cristian.rocha@moldeointeractive.com.ar> and others",
    ""
};

LIBVO_EXTERN(vdpaustereo)

#define CHECK_ST_ERROR(message) \
    if (vdp_st != VDP_STATUS_OK) { \
        mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] %s: %s\n", \
               message, vdp_get_error_string(vdp_st)); \
        return -1; \
    }

#define CHECK_ST_WARNING(message) \
    if (vdp_st != VDP_STATUS_OK) \
        mp_msg(MSGT_VO, MSGL_WARN, "[vdpaustereo] %s: %s\n", \
               message, vdp_get_error_string(vdp_st));

/* number of video and output surfaces */
#define NUM_OUTPUT_SURFACES                4
#define MAX_VIDEO_SURFACES                 50

/* number of palette entries */
#define PALETTE_SIZE 256

/* Initial maximum number of EOSD surfaces */
#define EOSD_SURFACES_INITIAL 512

/*
 * Global variable declaration - VDPAU specific
 */

/* Declaration for all variables of win_x11s_init_vdpau_procs() and
 * win_x11s_init_vdpau_flip_queue() functions
 */
static VdpDevice                           vdp_device;
static VdpGetProcAddress                  *vdp_get_proc_address;

struct VdpStream {
        VdpPresentationQueueTarget         vdp_flip_target;
        VdpPresentationQueue               vdp_flip_queue;
};

typedef struct VdpStream tVdpStream;

#define LEFT 0
#define RIGHT 1
static tVdpStream VdpStream[2];

#define MOLDEO_QUEUE(LEFT, RIGHT) ((frame_count & 0x1)? RIGHT: LEFT)
static unsigned int frame_count;

static VdpDeviceDestroy                   *vdp_device_destroy;
static VdpVideoSurfaceCreate              *vdp_video_surface_create;
static VdpVideoSurfaceDestroy             *vdp_video_surface_destroy;

static VdpGetErrorString                  *vdp_get_error_string;

/* May be used in software filtering/postprocessing options
 * in MPlayer (./mplayer -vf ..) if we copy video_surface data to
 * system memory.
 */
static VdpVideoSurfacePutBitsYCbCr        *vdp_video_surface_put_bits_y_cb_cr;
static VdpOutputSurfacePutBitsNative      *vdp_output_surface_put_bits_native;

static VdpOutputSurfaceCreate             *vdp_output_surface_create;
static VdpOutputSurfaceDestroy            *vdp_output_surface_destroy;

/* VideoMixer puts video_surface data on displayable output_surface. */
static VdpVideoMixerCreate                *vdp_video_mixer_create;
static VdpVideoMixerDestroy               *vdp_video_mixer_destroy;
static VdpVideoMixerRender                *vdp_video_mixer_render;
static VdpVideoMixerSetFeatureEnables     *vdp_video_mixer_set_feature_enables;
static VdpVideoMixerSetAttributeValues    *vdp_video_mixer_set_attribute_values;

static VdpPresentationQueueTargetDestroy *vdp_presentation_queue_target_destroy;
static VdpPresentationQueueCreate         *vdp_presentation_queue_create;
static VdpPresentationQueueDestroy        *vdp_presentation_queue_destroy;
static VdpPresentationQueueDisplay        *vdp_presentation_queue_display;
static VdpPresentationQueueBlockUntilSurfaceIdle *vdp_presentation_queue_block_until_surface_idle;
static VdpPresentationQueueTargetCreateX11        *vdp_presentation_queue_target_create_x11;

static VdpOutputSurfaceRenderOutputSurface       *vdp_output_surface_render_output_surface;
static VdpOutputSurfacePutBitsIndexed            *vdp_output_surface_put_bits_indexed;
static VdpOutputSurfaceRenderBitmapSurface       *vdp_output_surface_render_bitmap_surface;

static VdpBitmapSurfaceCreate                    *vdp_bitmap_surface_create;
static VdpBitmapSurfaceDestroy                   *vdp_bitmap_surface_destroy;
static VdpBitmapSurfacePutBitsNative             *vdp_bitmap_surface_putbits_native;

static VdpDecoderCreate                          *vdp_decoder_create;
static VdpDecoderDestroy                         *vdp_decoder_destroy;
static VdpDecoderRender                          *vdp_decoder_render;

static VdpGenerateCSCMatrix                      *vdp_generate_csc_matrix;
static VdpPreemptionCallbackRegister             *vdp_preemption_callback_register;

/* output_surfaces[NUM_OUTPUT_SURFACES] is misused for OSD. */
#define osd_surface output_surfaces[NUM_OUTPUT_SURFACES]
static VdpOutputSurface                   output_surfaces[NUM_OUTPUT_SURFACES + 1];
static VdpVideoSurface                    deint_surfaces[3];
static mp_image_t                        *deint_mpi[2];
static int                                output_surface_width, output_surface_height;

static VdpVideoMixer                      video_mixer;
static int                                deint;
static int                                deint_type;
static int                                deint_counter;
static int                                deint_buffer_past_frames;
static int                                pullup;
static float                              denoise;
static float                              sharpen;
static int                                colorspace;
static int                                chroma_deint;
static int                                force_mixer;
static int                                top_field_first;
static int                                flip;
static int                                hqscaling;

static VdpDecoder                         decoder;
static int                                decoder_max_refs;

static VdpRect                            src_rect_vid;
static VdpRect                            out_rect_vid;
static int                                border_x, border_y;

static struct vdpau_render_state          surface_render[MAX_VIDEO_SURFACES];

static int                                surface_num;

static int                                vid_surface_num;
static uint32_t                           vid_width, vid_height;
static uint32_t                           image_format;
static VdpChromaType                      vdp_chroma_type;
static VdpYCbCrFormat                     vdp_pixel_format;

static volatile int                       is_preempted;

/* draw_osd */
static unsigned char                     *index_data;
static int                                index_data_size;
static uint32_t                           palette[PALETTE_SIZE];
```

```
// EOSD
// Pool of surfaces
struct {
    VdpBitmapSurface surface;
    int w;
    int h;
    char in_use;
} *eosd_surfaces;

// List of surfaces to be rendered
struct {
    VdpBitmapSurface surface;
    VdpRect source;
    VdpRect dest;
    VdpColor color;
} *eosd_targets;

static int eosd_render_count;
static int eosd_surface_count;

// Video equalizer
static VdpProcamp procamp;

/*
 * X11 specific
 */
static int                                  visible_buf;
static int                                  int_pause;

static void flip_page_S(tVdpStream *VS);

static void draw_eosd(void);

static void push_deint_surface(VdpVideoSurface surface)
{
        deint_surfaces[2] = deint_surfaces[1];
        deint_surfaces[1] = deint_surfaces[0];
        deint_surfaces[0] = surface;
}

int setLR;

static void video_to_output_surface_S(tVdpStream *VS);

static void video_to_output_surface(void)
{
        video_to_output_surface_S(&VdpStream[frame_count & 0x1]);
}

static void video_to_output_surface_S(tVdpStream *VS)
{
    VdpTime dummy;
    VdpStatus vdp_st;
    int i;
    VdpPresentationQueue vdp_flip_queue = VS->vdp_flip_queue;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: video_to_output_surface {\n");

    if (vid_surface_num < 0)
        return;

    if (deint < 2 || deint_surfaces[0] == VDP_INVALID_HANDLE)
        push_deint_surface(surface_render[vid_surface_num].surface);

    for (i = 0; i <= !!(deint > 1); i++) {
        int field = VDP_VIDEO_MIXER_PICTURE_STRUCTURE_FRAME;
        VdpOutputSurface output_surface;

        if (i) {
            draw_eosd();
            draw_osd();
            flip_page_S(VS);
        }
        if (deint)
            field = (top_field_first == i) ^ (deint > 1) ?
                    VDP_VIDEO_MIXER_PICTURE_STRUCTURE_BOTTOM_FIELD:
                    VDP_VIDEO_MIXER_PICTURE_STRUCTURE_TOP_FIELD;

        output_surface = output_surfaces[surface_num];
        vdp_st = vdp_presentation_queue_block_until_surface_idle(vdp_flip_queue,
                                                                 output_surface,
                                                                 &dummy);
        CHECK_ST_WARNING("Error when calling vdp_presentation_queue_block_until_surface_idle")

        vdp_st = vdp_video_mixer_render(video_mixer, VDP_INVALID_HANDLE, 0,
                                        field, 2, deint_surfaces + 1,
                                        deint_surfaces[0],
                                        1, &surface_render[vid_surface_num].surface,
                                        &src_rect_vid,
                                        output_surface,
                                        NULL, &out_rect_vid, 0, NULL);
        CHECK_ST_WARNING("Error when calling vdp_video_mixer_render" )

        push_deint_surface(surface_render[vid_surface_num].surface);
    }
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: video_to_output_surface }\n");
}

static void resize(void)
{
    VdpStatus vdp_st;
    int i;
    struct vo_rect src_rect;
    struct vo_rect dst_rect;
    struct vo_rect borders;
    mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] resize (%i, %i) {\n", vid_width, vid_height);
    calc_src_dst_rects(vid_width, vid_height, &src_rect, &dst_rect, &borders, NULL);
    out_rect_vid.x0 = dst_rect.left;
    out_rect_vid.x1 = dst_rect.right;
    out_rect_vid.y0 = dst_rect.top;
    out_rect_vid.y1 = dst_rect.bottom;
    src_rect_vid.x0 = src_rect.left;
    src_rect_vid.x1 = src_rect.right;
    src_rect_vid.y0 = flip ? src_rect.bottom : src_rect.top;
    src_rect_vid.y1 = flip ? src_rect.top    : src_rect.bottom;
    border_x        = borders.left;
    border_y        = borders.top;
    mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] out_rect_vid (%i, %i, %i, %i) {\n", out_rect_vid.x0, out_rect_vid.x1, out_rec
t_vid.y0, out_rect_vid.y1 );
    mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] src_rect_vid (%i, %i, %i, %i) {\n", src_rect_vid.x0, src_rect_vid.x1, src_rec
t_vid.y0, src_rect_vid.y1 );
#ifdef CONFIG_FREETYPE
    // adjust font size to display size
    force_load_font = 1;
#endif
    vo_osd_changed(OSDTYPE_OSD);

    if (output_surface_width < vo_dwidth || output_surface_height < vo_dheight) {
        if (output_surface_width < vo_dwidth) {
            output_surface_width += output_surface_width >> 1;
            output_surface_width = FFMAX(output_surface_width, vo_dwidth);
        }
        if (output_surface_height < vo_dheight) {
            output_surface_height += output_surface_height >> 1;
            output_surface_height = FFMAX(output_surface_height, vo_dheight);
        }
        // Creation of output_surfaces
        for (i = 0; i <= NUM_OUTPUT_SURFACES; i++) {
            if (output_surfaces[i] != VDP_INVALID_HANDLE)
                vdp_output_surface_destroy(output_surfaces[i]);
            vdp_st = vdp_output_surface_create(vdp_device, VDP_RGBA_FORMAT_B8G8R8A8,
                                               output_surface_width, output_surface_height,
                                               &output_surfaces[i]);
            CHECK_ST_WARNING("Error when calling vdp_output_surface_create")
            mp_msg(MSGT_VO, MSGL_DBG2, "OUT CREATE: %u\n", output_surfaces[i]);
        }
    }
    if (image_format == IMGFMT_BGRA) {
        vdp_st = vdp_output_surface_render_output_surface(output_surfaces[surface_num],
                                                          NULL, VDP_INVALID_HANDLE,
                                                          NULL, NULL, NULL,
                                                          VDP_OUTPUT_SURFACE_RENDER_ROTATE_0);
        CHECK_ST_WARNING("Error when calling vdp_output_surface_render_output_surface")
        vdp_st = vdp_output_surface_render_output_surface(output_surfaces[1 - surface_num],
                                                          NULL, VDP_INVALID_HANDLE,
                                                          NULL, NULL, NULL,
                                                          VDP_OUTPUT_SURFACE_RENDER_ROTATE_0);
        CHECK_ST_WARNING("Error when calling vdp_output_surface_render_output_surface")
    } else
        video_to_output_surface();
    if (visible_buf)
        flip_page();
    mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] resize }\n");
}

static void preemption_callback(VdpDevice device, void *context)
{
    mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] Display preemption detected\n");
    is_preempted = 1;
}

/* Initialize vdp_get_proc_address, called from preinit() */
static int win_x11s_init_vdpau_procs(void)
{
    VdpStatus vdp_st;

    struct vdp_function {
        const int id;
        void *pointer;
    };

    const struct vdp_function *dsc;

    static const struct vdp_function vdp_func[] = {
        {VDP_FUNC_ID_GET_ERROR_STRING,          &vdp_get_error_string},
        {VDP_FUNC_ID_DEVICE_DESTROY,            &vdp_device_destroy},
        {VDP_FUNC_ID_VIDEO_SURFACE_CREATE,      &vdp_video_surface_create},
        {VDP_FUNC_ID_VIDEO_SURFACE_DESTROY,     &vdp_video_surface_destroy},
        {VDP_FUNC_ID_VIDEO_SURFACE_PUT_BITS_Y_CB_CR,
                        &vdp_video_surface_put_bits_y_cb_cr},
        {VDP_FUNC_ID_OUTPUT_SURFACE_PUT_BITS_NATIVE,
                        &vdp_output_surface_put_bits_native},
        {VDP_FUNC_ID_OUTPUT_SURFACE_CREATE,     &vdp_output_surface_create},
        {VDP_FUNC_ID_OUTPUT_SURFACE_DESTROY,    &vdp_output_surface_destroy},
        {VDP_FUNC_ID_VIDEO_MIXER_CREATE, &vdp_video_mixer_create},
        {VDP_FUNC_ID_VIDEO_MIXER_DESTROY,       &vdp_video_mixer_destroy},
        {VDP_FUNC_ID_VIDEO_MIXER_RENDER,        &vdp_video_mixer_render},
        {VDP_FUNC_ID_VIDEO_MIXER_SET_FEATURE_ENABLES,
                        &vdp_video_mixer_set_feature_enables},
```

```
        {VDP_FUNC_ID_VIDEO_MIXER_SET_ATTRIBUTE_VALUES,
                        &vdp_video_mixer_set_attribute_values},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_TARGET_DESTROY,
                        &vdp_presentation_queue_target_destroy},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_CREATE, &vdp_presentation_queue_create},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_DESTROY,
                        &vdp_presentation_queue_destroy},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_DISPLAY,
                        &vdp_presentation_queue_display},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_BLOCK_UNTIL_SURFACE_IDLE,
                        &vdp_presentation_queue_block_until_surface_idle},
        {VDP_FUNC_ID_PRESENTATION_QUEUE_TARGET_CREATE_X11,
                        &vdp_presentation_queue_target_create_x11},
        {VDP_FUNC_ID_OUTPUT_SURFACE_RENDER_OUTPUT_SURFACE,
                        &vdp_output_surface_render_output_surface},
        {VDP_FUNC_ID_OUTPUT_SURFACE_PUT_BITS_INDEXED,
                        &vdp_output_surface_put_bits_indexed},
        {VDP_FUNC_ID_DECODER_CREATE,            &vdp_decoder_create},
        {VDP_FUNC_ID_DECODER_RENDER,            &vdp_decoder_render},
        {VDP_FUNC_ID_DECODER_DESTROY,           &vdp_decoder_destroy},
        {VDP_FUNC_ID_BITMAP_SURFACE_CREATE,     &vdp_bitmap_surface_create},
        {VDP_FUNC_ID_BITMAP_SURFACE_DESTROY,    &vdp_bitmap_surface_destroy},
        {VDP_FUNC_ID_BITMAP_SURFACE_PUT_BITS_NATIVE,
                        &vdp_bitmap_surface_putbits_native},
        {VDP_FUNC_ID_OUTPUT_SURFACE_RENDER_BITMAP_SURFACE,
                        &vdp_output_surface_render_bitmap_surface},
        {VDP_FUNC_ID_GENERATE_CSC_MATRIX,       &vdp_generate_csc_matrix},
        {VDP_FUNC_ID_PREEMPTION_CALLBACK_REGISTER,
                        &vdp_preemption_callback_register},
        {0, NULL}
    };
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: win_x11s_init_vdpau_procs {\n");

    vdp_st = vdp_device_create_x11(ScreenLeft.display, ScreenLeft.screen,
                                   &vdp_device, &vdp_get_proc_address);
    if (vdp_st != VDP_STATUS_OK) {
        mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] Error when calling vdp_device_create_x11: %i\n", vdp_st);
        return -1;
    }

    vdp_get_error_string = NULL;
    for (dsc = vdp_func; dsc->pointer; dsc++) {
        vdp_st = vdp_get_proc_address(vdp_device, dsc->id, dsc->pointer);
        if (vdp_st != VDP_STATUS_OK) {
            mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] Error when calling vdp_get_proc_address(function id %d): %s\n",
                   dsc->id, vdp_get_error_string ? vdp_get_error_string(vdp_st) : "?");
            return -1;
        }
    }
    vdp_st = vdp_preemption_callback_register(vdp_device,
                                              preemption_callback, NULL);
    CHECK_ST_ERROR("Error when calling vdp_preemption_callback_register")

    return 0;
}

// TODO: Revisar con detenimiento
static int win_x11s_init_vdpau_flip_queue_S(tScreenOutput *Screen, tVdpStream *VdpStream);

static int win_x11s_init_vdpau_flip_queue(void)
{
        int r;
        mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: win_x11s_init_vdpau_flip_queue {\n");
        r = win_x11s_init_vdpau_flip_queue_S(&ScreenLeft, &VdpStream[LEFT]);
        r = win_x11s_init_vdpau_flip_queue_S(&ScreenRight, &VdpStream[RIGHT]);
        mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: win_x11s_init_vdpau_flip_queue }\n");
        return r;
}

static int win_x11s_init_vdpau_flip_queue_S(tScreenOutput *Screen, tVdpStream *VS)
{
    VdpStatus vdp_st;

    vdp_st = vdp_presentation_queue_target_create_x11(vdp_device, Screen->window,
                                                      &VS->vdp_flip_target);
    CHECK_ST_ERROR("Error when calling vdp_presentation_queue_target_create_x11")

    vdp_st = vdp_presentation_queue_create(vdp_device, VS->vdp_flip_target,
                                           &VS->vdp_flip_queue);
    CHECK_ST_ERROR("Error when calling vdp_presentation_queue_create")

    return 0;
}

static int update_csc_matrix(void)
{
    VdpStatus vdp_st;
    VdpCSCMatrix matrix;
    static const VdpVideoMixerAttribute attributes[] = {VDP_VIDEO_MIXER_ATTRIBUTE_CSC_MATRIX};
    const void *attribute_values[] = {&matrix};
    static const VdpColorStandard vdp_colors[] = {0, VDP_COLOR_STANDARD_ITUR_BT_601, VDP_COLOR_STANDARD_ITU
R_BT_709, VDP_COLOR_STANDARD_SMPTE_240M};
    static const char * const vdp_names[] = {NULL, "BT.601", "BT.709", "SMPTE-240M"};
    int csp = colorspace;

    if (!csp)
        csp = vid_width >= 1280 || vid_height > 576 ? 2 : 1;

    mp_msg(MSGT_VO, MSGL_V, "[vdpaustereo] Updating CSC matrix for %s\n",
           vdp_names[csp]);

    vdp_st = vdp_generate_csc_matrix(&procamp, vdp_colors[csp], &matrix);
    CHECK_ST_WARNING("Error when generating CSC matrix")

    vdp_st = vdp_video_mixer_set_attribute_values(video_mixer, 1, attributes,
                                                  attribute_values);
    CHECK_ST_WARNING("Error when setting CSC matrix")
    return VO_TRUE;
}

static int create_vdp_mixer(VdpChromaType vdp_chroma_type)
{
#define VDP_NUM_MIXER_PARAMETER 3
#define MAX_NUM_FEATURES 6
    int i;
    VdpStatus vdp_st;
    int feature_count = 0;
    VdpVideoMixerFeature features[MAX_NUM_FEATURES];
    VdpBool feature_enables[MAX_NUM_FEATURES];
    static const VdpVideoMixerAttribute denoise_attrib[] = {VDP_VIDEO_MIXER_ATTRIBUTE_NOISE_REDUCTION_LEVEL
};
    const void * const denoise_value[] = {&denoise};
    static const VdpVideoMixerAttribute sharpen_attrib[] = {VDP_VIDEO_MIXER_ATTRIBUTE_SHARPNESS_LEVEL};
    const void * const sharpen_value[] = {&sharpen};
    static const VdpVideoMixerAttribute skip_chroma_attrib[] = {VDP_VIDEO_MIXER_ATTRIBUTE_SKIP_CHROMA_DEINT
ERLACE};
    const uint8_t skip_chroma_value = 1;
    const void * const skip_chroma_value_ptr[] = {&skip_chroma_value};
    static const VdpVideoMixerParameter parameters[VDP_NUM_MIXER_PARAMETER] = {
        VDP_VIDEO_MIXER_PARAMETER_VIDEO_SURFACE_WIDTH,
        VDP_VIDEO_MIXER_PARAMETER_VIDEO_SURFACE_HEIGHT,
        VDP_VIDEO_MIXER_PARAMETER_CHROMA_TYPE
    };
    const void *const parameter_values[VDP_NUM_MIXER_PARAMETER] = {
        &vid_width,
        &vid_height,
        &vdp_chroma_type
    };
    features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_DEINTERLACE_TEMPORAL;
    if (deint == 4)
        features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_DEINTERLACE_TEMPORAL_SPATIAL;
    if (pullup)
        features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_INVERSE_TELECINE;
    if (denoise)
        features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_NOISE_REDUCTION;
    if (sharpen)
        features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_SHARPNESS;
    if (hqscaling)
        features[feature_count++] = VDP_VIDEO_MIXER_FEATURE_HIGH_QUALITY_SCALING_L1 + (hqscaling - 1);

    vdp_st = vdp_video_mixer_create(vdp_device, feature_count, features,
                                    VDP_NUM_MIXER_PARAMETER,
                                    parameters, parameter_values,
                                    &video_mixer);
    CHECK_ST_ERROR("Error when calling vdp_video_mixer_create")

    for (i = 0; i < feature_count; i++)
        feature_enables[i] = VDP_TRUE;
    if (deint < 3)
        feature_enables[0] = VDP_FALSE;
    if (feature_count)
        vdp_video_mixer_set_feature_enables(video_mixer, feature_count, features, feature_enables);
    if (denoise)
        vdp_video_mixer_set_attribute_values(video_mixer, 1, denoise_attrib, denoise_value);
    if (sharpen)
        vdp_video_mixer_set_attribute_values(video_mixer, 1, sharpen_attrib, sharpen_value);
    if (!chroma_deint)
        vdp_video_mixer_set_attribute_values(video_mixer, 1, skip_chroma_attrib, skip_chroma_value_ptr);

    update_csc_matrix();
    return 0;
}

// Free everything specific to a certain video file
static void free_video_specific(void)
{
    int i;
    VdpStatus vdp_st;

    if (decoder != VDP_INVALID_HANDLE)
        vdp_decoder_destroy(decoder);
    decoder = VDP_INVALID_HANDLE;
    decoder_max_refs = -1;

    for (i = 0; i < 3; i++)
        deint_surfaces[i] = VDP_INVALID_HANDLE;

    for (i = 0; i < 2; i++)
        if (deint_mpi[i]) {
            deint_mpi[i]->usage_count--;
            deint_mpi[i] = NULL;
        }

    for (i = 0; i < MAX_VIDEO_SURFACES; i++) {
        if (surface_render[i].surface != VDP_INVALID_HANDLE) {
          vdp_st = vdp_video_surface_destroy(surface_render[i].surface);
          CHECK_ST_WARNING("Error when calling vdp_video_surface_destroy")
        }
        surface_render[i].surface = VDP_INVALID_HANDLE;
    }
```

```
    if (video_mixer != VDP_INVALID_HANDLE) {
        vdp_st = vdp_video_mixer_destroy(video_mixer);
        CHECK_ST_WARNING("Error when calling vdp_video_mixer_destroy")
    }
    video_mixer = VDP_INVALID_HANDLE;
}

static int create_vdp_decoder(uint32_t format, uint32_t width, uint32_t height,
                              int max_refs)
{
    VdpStatus vdp_st;
    VdpDecoderProfile vdp_decoder_profile;
    if (decoder != VDP_INVALID_HANDLE)
        vdp_decoder_destroy(decoder);
    switch (format) {
    case IMGFMT_VDPAU_MPEG1:
        vdp_decoder_profile = VDP_DECODER_PROFILE_MPEG1;
        break;
    case IMGFMT_VDPAU_MPEG2:
        vdp_decoder_profile = VDP_DECODER_PROFILE_MPEG2_MAIN;
        break;
    case IMGFMT_VDPAU_H264:
        vdp_decoder_profile = VDP_DECODER_PROFILE_H264_HIGH;
        mp_msg(MSGT_VO, MSGL_V, "[vdpaustereo] Creating H264 hardware decoder for %d reference frames.\n", max_refs);
        break;
    case IMGFMT_VDPAU_WMV3:
        vdp_decoder_profile = VDP_DECODER_PROFILE_VC1_MAIN;
        break;
    case IMGFMT_VDPAU_VC1:
        vdp_decoder_profile = VDP_DECODER_PROFILE_VC1_ADVANCED;
        break;
    case IMGFMT_VDPAU_MPEG4:
        vdp_decoder_profile = VDP_DECODER_PROFILE_MPEG4_PART2_ASP;
        break;
    default:
        goto err_out;
    }
    vdp_st = vdp_decoder_create(vdp_device, vdp_decoder_profile,
                                width, height, max_refs, &decoder);
    CHECK_ST_WARNING("Failed creating VDPAU decoder");
    if (vdp_st != VDP_STATUS_OK) {
err_out:
        decoder = VDP_INVALID_HANDLE;
        decoder_max_refs = 0;
        return 0;
    }
    decoder_max_refs = max_refs;
    return 1;
}

// Sin pasar a STEREOSCOPIC
static void mark_vdpau_objects_uninitialized(void)
{
    int i;

    decoder = VDP_INVALID_HANDLE;
    for (i = 0; i < MAX_VIDEO_SURFACES; i++)
        surface_render[i].surface = VDP_INVALID_HANDLE;
    for (i = 0; i < 3; i++) {
        deint_surfaces[i] = VDP_INVALID_HANDLE;
        if (i < 2 && deint_mpi[i])
            deint_mpi[i]->usage_count--;
        deint_mpi[i] = NULL;
    }
    video_mixer     = VDP_INVALID_HANDLE;
    VdpStream[LEFT].vdp_flip_queue  = VDP_INVALID_HANDLE;
    VdpStream[LEFT].vdp_flip_target = VDP_INVALID_HANDLE;
    VdpStream[RIGHT].vdp_flip_queue  = VDP_INVALID_HANDLE;
    VdpStream[RIGHT].vdp_flip_target = VDP_INVALID_HANDLE;
    for (i = 0; i <= NUM_OUTPUT_SURFACES; i++)
        output_surfaces[i] = VDP_INVALID_HANDLE;
    vdp_device = VDP_INVALID_HANDLE;
    for (i = 0; i < eosd_surface_count; i++)
        eosd_surfaces[i].surface = VDP_INVALID_HANDLE;
    output_surface_width = output_surface_height = -1;
    eosd_render_count = 0;
    visible_buf = 0;
}

static int handle_preemption(void)
{
    if (!is_preempted)
        return 0;
    is_preempted = 0;
    mp_msg(MSGT_VO, MSGL_INFO, "[vdpaustereo] Attempting to recover from preemption.\n");
    mark_vdpau_objects_uninitialized();
    if (win_x11s_init_vdpau_procs() < 0 ||
        win_x11s_init_vdpau_flip_queue() < 0 ||
        create_vdp_mixer(vdp_chroma_type) < 0) {
        mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] Recovering from preemption failed\n");
        is_preempted = 1;
        return -1;
    }
    resize();
    mp_msg(MSGT_VO, MSGL_INFO, "[vdpaustereo] Recovered from display preemption.\n");
    return 1;
}

/*
 * connect to X server, create and map window, initialize all
 * VDPAU objects, create different surfaces etc.
 */
static int config(uint32_t width, uint32_t height, uint32_t d_width,
                  uint32_t d_height, uint32_t flags, char *title,
                  uint32_t format)
{
    XVisualInfo vinfo;
    XSetWindowAttributes xswa;
    XWindowAttributes attribs;
    unsigned long xswamask;
    int depth;

#ifdef CONFIG_XF86VM
    int vm = flags & VOFLAG_MODESWITCHING;
#endif
    flip = flags & VOFLAG_FLIPPING;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: config(%i,%i,%i,%i...) {\n", width, height, d_width, d_height);

    image_format = format;
    vid_width    = width;
    vid_height   = height;
    free_video_specific();
    if (IMGFMT_IS_VDPAU(image_format)
        && !create_vdp_decoder(image_format, vid_width, vid_height, 2)) // Two reference frames
        return -1;

    int_pause   = 0;
    visible_buf = 0;

#ifdef CONFIG_GUI
    if (use_gui)
        guiGetEvent(guiSetShVideo, 0);  // the GUI will set up / resize our window
    else
#endif
    {
#ifdef CONFIG_XF86VM
        if (vm)
            vo_vm_switch();
        else
#endif
        XGetWindowAttributes(ScreenLeft.display, DefaultRootWindow(ScreenLeft.display), &attribs);
        depth = attribs.depth;
        if (depth != 15 && depth != 16 && depth != 24 && depth != 32)
            depth = 24;
        XMatchVisualInfo(ScreenLeft.display, ScreenLeft.screen, depth, TrueColor, &vinfo);
        xswa.background_pixel = 0;
        xswa.border_pixel     = 0;
        /* Do not use CWBackPixel: It leads to VDPAU errors after
           aspect ratio changes. */
        xswamask = CWBorderPixel;

        vo_x11s_create_vo_window_S(&ScreenLeft, vo_dx, vo_dy, d_width, d_height,
                                   flags, CopyFromParent, "vdpaustereo", title);
        XChangeWindowAttributes(ScreenLeft.display, ScreenLeft.window, xswamask, &xswa);
        vo_x11s_create_vo_window_S(&ScreenRight, vo_dx, vo_dy, d_width, d_height,
                                   flags, CopyFromParent, "vdpaustereo", title);
        XChangeWindowAttributes(ScreenRight.display, ScreenRight.window, xswamask, &xswa);

#ifdef CONFIG_XF86VM
        if (vm) {
            if (vo_grabpointer)
                XGrabPointer(ScreenLeft.display, ScreenLeft.window, True, 0,
                             GrabModeAsync, GrabModeAsync,
                             ScreenLeft.window, None, CurrentTime);
            XSetInputFocus(ScreenLeft.display, ScreenLeft.window, RevertToNone, CurrentTime);

            if (vo_grabpointer)
                XGrabPointer(ScreenRight.display, ScreenRight.window, True, 0,
                             GrabModeAsync, GrabModeAsync,
                             ScreenRight.window, None, CurrentTime);
            XSetInputFocus(ScreenRight.display, ScreenRight.window, RevertToNone, CurrentTime);
        }
#endif
    }

    if ((flags & VOFLAG_FULLSCREEN) && WinID <= 0)
        vo_fs = 0;

    // -----VDPAU related code here --------
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: Setup VdpStream {\n");
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] LEFT\n");
    if (VdpStream[LEFT].vdp_flip_queue == VDP_INVALID_HANDLE &&
                win_x11s_init_vdpau_flip_queue_S(&ScreenLeft, &VdpStream[LEFT]))
        return -1;
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] RIGHT\n");
    if (VdpStream[RIGHT].vdp_flip_queue == VDP_INVALID_HANDLE &&
                win_x11s_init_vdpau_flip_queue_S(&ScreenRight, &VdpStream[RIGHT]))
        return -1;
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: Setup VdpStream }\n");

    vdp_chroma_type = VDP_CHROMA_TYPE_420;
    switch (image_format) {
    case IMGFMT_YV12:
    case IMGFMT_I420:
    case IMGFMT_IYUV:
        vdp_pixel_format = VDP_YCBCR_FORMAT_YV12;
        break;
    case IMGFMT_NV12:
```

```
        vdp_pixel_format = VDP_YCBCR_FORMAT_NV12;
        break;
    case IMGFMT_YUY2:
        vdp_pixel_format = VDP_YCBCR_FORMAT_YUYV;
        vdp_chroma_type  = VDP_CHROMA_TYPE_422;
        break;
    case IMGFMT_UYVY:
        vdp_pixel_format = VDP_YCBCR_FORMAT_UYVY;
        vdp_chroma_type  = VDP_CHROMA_TYPE_422;
    }
    if (create_vdp_mixer(vdp_chroma_type))
        return -1;

    surface_num     =  0;
    vid_surface_num = -1;
    resize();

    frame_count = 0;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: config }\n");
    return 0;
}

static void check_events_S(tScreenOutput *Screen, tVdpStream *VS);

static void check_events(void)
{
        check_events_S(&ScreenLeft, &VdpStream[LEFT]);
        check_events_S(&ScreenRight, &VdpStream[RIGHT]);
}

static void check_events_S(tScreenOutput *Screen, tVdpStream *VS)
{
    VdpStatus vdp_st;
    int e = vo_x11s_check_events_S(Screen);

    if (handle_preemption() < 0)
        return;

    if (e & VO_EVENT_RESIZE)
        resize();

    if ((e & VO_EVENT_EXPOSE || e & VO_EVENT_RESIZE) && int_pause) {
        // did we already draw a buffer
        if (visible_buf) {
            mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] EXPOSE %i %i \n", frame_count, surface_num);

            // redraw the last visible buffer
            vdp_st = vdp_presentation_queue_display(VS->vdp_flip_queue,
                                                    output_surfaces[surface_num],
                                                    vo_dwidth, vo_dheight,
                                                    0);
            CHECK_ST_WARNING("Error when calling vdp_presentation_queue_display")
        }
    }
}

static void draw_osd_I8A8(int x0,int y0, int w,int h, unsigned char *src,
                          unsigned char *srca, int stride)
{
    VdpOutputSurface output_surface = output_surfaces[surface_num];
    VdpStatus vdp_st;
    int i, j;
    int pitch;
    int index_data_size_required;
    VdpRect output_indexed_rect_vid;
    VdpOutputSurfaceRenderBlendState blend_state;

    if (!w || !h)
        return;

    index_data_size_required = 2*w*h;
    if (index_data_size < index_data_size_required) {
        index_data      = realloc(index_data, index_data_size_required);
        index_data_size = index_data_size_required;
    }

    // index_data creation, component order - I, A, I, A, .....
    for (i = 0; i < h; i++)
        for (j = 0; j < w; j++) {
            index_data[i*2*w + j*2]     =  src [i*stride + j];
            index_data[i*2*w + j*2 + 1] = -srca[i*stride + j];
        }

    output_indexed_rect_vid.x0 = x0;
    output_indexed_rect_vid.y0 = y0;
    output_indexed_rect_vid.x1 = x0 + w;
    output_indexed_rect_vid.y1 = y0 + h;

    pitch = w*2;

    // write source_data to osd_surface.
    vdp_st = vdp_output_surface_put_bits_indexed(osd_surface,
                                                 VDP_INDEXED_FORMAT_I8A8,
                                                 (const void *const*)&index_data,
                                                 &pitch,
                                                 &output_indexed_rect_vid,
                                                 VDP_COLOR_TABLE_FORMAT_B8G8R8X8,
                                                 (void *)palette);
    CHECK_ST_WARNING("Error when calling vdp_output_surface_put_bits_indexed")

    blend_state.struct_version                = VDP_OUTPUT_SURFACE_RENDER_BLEND_STATE_VERSION;
    blend_state.blend_factor_source_color     = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE;
    blend_state.blend_factor_source_alpha     = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE;
    blend_state.blend_factor_destination_color = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA
;
    blend_state.blend_factor_destination_alpha = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA
;
    blend_state.blend_equation_color          = VDP_OUTPUT_SURFACE_RENDER_BLEND_EQUATION_ADD;
    blend_state.blend_equation_alpha          = VDP_OUTPUT_SURFACE_RENDER_BLEND_EQUATION_ADD;

    vdp_st = vdp_output_surface_render_output_surface(output_surface,
                                                       &output_indexed_rect_vid,
                                                       osd_surface,
                                                       &output_indexed_rect_vid,
                                                       NULL,
                                                       &blend_state,
                                                       VDP_OUTPUT_SURFACE_RENDER_ROTATE_0);
    CHECK_ST_WARNING("Error when calling vdp_output_surface_render_output_surface")
}

static void draw_eosd(void)
{
    VdpStatus vdp_st;
    VdpOutputSurface output_surface = output_surfaces[surface_num];
    VdpOutputSurfaceRenderBlendState blend_state;
    int i;

    blend_state.struct_version                = VDP_OUTPUT_SURFACE_RENDER_BLEND_STATE_VERSION;
    blend_state.blend_factor_source_color     = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_SRC_ALPHA;
    blend_state.blend_factor_source_alpha     = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE;
    blend_state.blend_factor_destination_color = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA
;
    blend_state.blend_factor_destination_alpha = VDP_OUTPUT_SURFACE_RENDER_BLEND_FACTOR_SRC_ALPHA;
    blend_state.blend_equation_color          = VDP_OUTPUT_SURFACE_RENDER_BLEND_EQUATION_ADD;
    blend_state.blend_equation_alpha          = VDP_OUTPUT_SURFACE_RENDER_BLEND_EQUATION_ADD;

    for (i = 0; i < eosd_render_count; i++) {
        vdp_st = vdp_output_surface_render_bitmap_surface(
            output_surface, &eosd_targets[i].dest,
            eosd_targets[i].surface, &eosd_targets[i].source,
            &eosd_targets[i].color, &blend_state,
            VDP_OUTPUT_SURFACE_RENDER_ROTATE_0);
        CHECK_ST_WARNING("EOSD: Error when rendering")
    }
}

static void generate_eosd(mp_eosd_images_t *imgs)
{
    VdpStatus vdp_st;
    VdpRect destRect;
    int j, found;
    ass_image_t *img = imgs->imgs;
    ass_image_t *i;

    // Nothing changed, no need to redraw
    if (imgs->changed == 0)
        return;
    eosd_render_count = 0;
    // There's nothing to render!
    if (!img)
        return;

    if (imgs->changed == 1)
        goto eosd_skip_upload;

    for (j = 0; j < eosd_surface_count; j++)
        eosd_surfaces[j].in_use = 0;

    for (i = img; i; i = i->next) {
        // Try to reuse a suitable surface
        found = -1;
        for (j = 0; j < eosd_surface_count; j++) {
            if (eosd_surfaces[j].surface != VDP_INVALID_HANDLE && !eosd_surfaces[j].in_use &&
                eosd_surfaces[j].w >= i->w && eosd_surfaces[j].h >= i->h) {
                found = j;
                break;
            }
        }
        // None found, allocate a new surface
        if (found < 0) {
            for (j = 0; j < eosd_surface_count; j++) {
                if (!eosd_surfaces[j].in_use) {
                    if (eosd_surfaces[j].surface != VDP_INVALID_HANDLE)
                        vdp_bitmap_surface_destroy(eosd_surfaces[j].surface);
                    found = j;
                    break;
                }
            }
            // Allocate new space for surface/target arrays
            if (found < 0) {
                j = found = eosd_surface_count;
                eosd_surface_count = eosd_surface_count ? eosd_surface_count*2 : EOSD_SURFACES_INITIAL;
                eosd_surfaces = realloc(eosd_surfaces, eosd_surface_count * sizeof(*eosd_surfaces));
                eosd_targets  = realloc(eosd_targets,  eosd_surface_count * sizeof(*eosd_targets));
                for (j = found; j < eosd_surface_count; j++) {
                    eosd_surfaces[j].surface = VDP_INVALID_HANDLE;
                    eosd_surfaces[j].in_use = 0;
                }
            }
```

```
            vdp_st = vdp_bitmap_surface_create(vdp_device, VDP_RGBA_FORMAT_A8,
                i->w, i->h, VDP_TRUE, &eosd_surfaces[found].surface);
            CHECK_ST_WARNING("EOSD: error when creating surface")
            eosd_surfaces[found].w = i->w;
            eosd_surfaces[found].h = i->h;
        }
        eosd_surfaces[found].in_use = 1;
        eosd_targets[eosd_render_count].surface = eosd_surfaces[found].surface;
        destRect.x0 = 0;
        destRect.y0 = 0;
        destRect.x1 = i->w;
        destRect.y1 = i->h;
        vdp_st = vdp_bitmap_surface_putbits_native(eosd_targets[eosd_render_count].surface,
            (const void *) &i->bitmap, &i->stride, &destRect);
        CHECK_ST_WARNING("EOSD: putbits failed")
        eosd_render_count++;
    }

eosd_skip_upload:
    eosd_render_count = 0;
    for (i = img; i; i = i->next) {
        // Render dest, color, etc.
        eosd_targets[eosd_render_count].color.alpha = 1.0 - ((i->color >>  0) & 0xff) / 255.0;
        eosd_targets[eosd_render_count].color.blue  =       ((i->color >>  8) & 0xff) / 255.0;
        eosd_targets[eosd_render_count].color.green =       ((i->color >> 16) & 0xff) / 255.0;
        eosd_targets[eosd_render_count].color.red   =       ((i->color >> 24) & 0xff) / 255.0;
        eosd_targets[eosd_render_count].dest.x0     =        i->dst_x;
        eosd_targets[eosd_render_count].dest.y0     =        i->dst_y;
        eosd_targets[eosd_render_count].dest.x1     = i->w + i->dst_x;
        eosd_targets[eosd_render_count].dest.y1     = i->h + i->dst_y;
        eosd_targets[eosd_render_count].source.x0   = 0;
        eosd_targets[eosd_render_count].source.y0   = 0;
        eosd_targets[eosd_render_count].source.x1   = i->w;
        eosd_targets[eosd_render_count].source.y1   = i->h;
        eosd_render_count++;
    }
}

static void draw_osd(void)
{
    mp_msg(MSGT_VO, MSGL_DBG2, "DRAW_OSD\n");

    if (handle_preemption() < 0)
        return;

    vo_draw_text_ext(vo_dwidth, vo_dheight, border_x, border_y, border_x, border_y,
                     vid_width, vid_height, draw_osd_I8A8);
}


static void flip_page(void)
{
        flip_page_S(&VdpStream[frame_count & 0x01]);
}

static void flip_page_S(tVdpStream *VS)
{
    VdpStatus vdp_st;
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: flip_page {\n");

    mp_msg(MSGT_VO, MSGL_DBG2, "\nFLIP_PAGE VID:%u -> OUT:%u (%i)\n",
           surface_render[vid_surface_num].surface, output_surfaces[surface_num], frame_count);

    if (handle_preemption() < 0)
        return;

    vdp_st = vdp_presentation_queue_display(VS->vdp_flip_queue, output_surfaces[surface_num],
                                            vo_dwidth, vo_dheight,
                                            0);
    CHECK_ST_WARNING("Error when calling vdp_presentation_queue_display")

    surface_num = (surface_num + 1) % NUM_OUTPUT_SURFACES;
    visible_buf = 1;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: flip_page }\n");
}

static int draw_slice(uint8_t *image[], int stride[], int w, int h,
                      int x, int y)
{
    VdpStatus vdp_st;
    struct vdpau_render_state *rndr = (struct vdpau_render_state *)image[0];
    int max_refs = image_format == IMGFMT_VDPAU_H264 ? rndr->info.h264.num_ref_frames : 2;

    if (handle_preemption() < 0)
        return VO_TRUE;

    if (!IMGFMT_IS_VDPAU(image_format))
        return VO_FALSE;
    if ((decoder == VDP_INVALID_HANDLE || decoder_max_refs < max_refs)
        && !create_vdp_decoder(image_format, vid_width, vid_height, max_refs))
        return VO_FALSE;

    vdp_st = vdp_decoder_render(decoder, rndr->surface, (void *)&rndr->info, rndr->bitstream_buffers_used,
rndr->bitstream_buffers);
    CHECK_ST_WARNING("Failed VDPAU decoder rendering");
    return VO_TRUE;
}
```

```
static int draw_frame(uint8_t *src[])
{
    return VO_ERROR;
}

static struct vdpau_render_state *get_surface(int number)
{
    if (number > MAX_VIDEO_SURFACES)
        return NULL;
    if (surface_render[number].surface == VDP_INVALID_HANDLE) {
        VdpStatus vdp_st;
        vdp_st = vdp_video_surface_create(vdp_device, vdp_chroma_type,
                                          vid_width, vid_height,
                                          &surface_render[number].surface);
        CHECK_ST_WARNING("Error when calling vdp_video_surface_create")
        if (vdp_st != VDP_STATUS_OK)
            return NULL;
    }
    mp_msg(MSGT_VO, MSGL_DBG2, "VID CREATE: %u\n", surface_render[number].surface);
    return &surface_render[number];
}

static uint32_t draw_image(mp_image_t *mpi)
{
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: draw_image (frame: %i) {\n" , frame_count);

    if (IMGFMT_IS_VDPAU(image_format)) {
        struct vdpau_render_state *rndr = mpi->priv;
        vid_surface_num = rndr - surface_render;
        if (deint_buffer_past_frames) {
            mpi->usage_count++;
            if (deint_mpi[1])
                deint_mpi[1]->usage_count--;
            deint_mpi[1] = deint_mpi[0];
            deint_mpi[0] = mpi;
        }
    } else if (image_format == IMGFMT_BGRA) {
        VdpStatus vdp_st;
        VdpRect r = {0, 0, vid_width, vid_height};
        vdp_st = vdp_output_surface_put_bits_native(output_surfaces[2],
                                                    (void const*const*)mpi->planes,
                                                    mpi->stride, &r);
        CHECK_ST_ERROR("Error when calling vdp_output_surface_put_bits_native")
        vdp_st = vdp_output_surface_render_output_surface(output_surfaces[surface_num],
                                                          &out_rect_vid,
                                                          output_surfaces[2],
                                                          &src_rect_vid, NULL, NULL,
                                                          VDP_OUTPUT_SURFACE_RENDER_ROTATE_0);
        CHECK_ST_ERROR("Error when calling vdp_output_surface_render_output_surface")
    } else if (!(mpi->flags & MP_IMGFLAG_DRAW_CALLBACK)) {
        VdpStatus vdp_st;
        void *destdata[3] = {mpi->planes[0], mpi->planes[2], mpi->planes[1]};
        struct vdpau_render_state *rndr = get_surface(deint_counter);
        deint_counter = (deint_counter + 1) % 3;
        vid_surface_num = rndr - surface_render;
        if (image_format == IMGFMT_NV12)
            destdata[1] = destdata[2];
        vdp_st = vdp_video_surface_put_bits_y_cb_cr(rndr->surface,
                                                    vdp_pixel_format,
                                                    (const void *const*)destdata,
                                                    mpi->stride); // pitch
        CHECK_ST_ERROR("Error when calling vdp_video_surface_put_bits_y_cb_cr")
    }
    if (mpi->fields & MP_IMGFIELD_ORDERED)
        top_field_first = !!(mpi->fields & MP_IMGFIELD_TOP_FIRST);
    else
        top_field_first = 1;

    video_to_output_surface_S(&VdpStream[frame_count & 0x1]);

    frame_count++;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: draw_image }\n");
    return VO_TRUE;
}

static uint32_t get_image(mp_image_t *mpi)
{
    struct vdpau_render_state *rndr;
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: get_image {\n");

    // no dr for non-decoding for now
    if (!IMGFMT_IS_VDPAU(image_format))
        return VO_FALSE;
    if (mpi->type != MP_IMGTYPE_NUMBERED)
        return VO_FALSE;

    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: mpi->number %i\n", mpi->number);
    rndr = get_surface(mpi->number);
    if (!rndr) {
        mp_msg(MSGT_VO, MSGL_ERR, "[vdpaustereo] no surfaces available in get_image\n");
        // TODO: this probably breaks things forever, provide a dummy buffer?
        return VO_FALSE;
    }
    mpi->flags |= MP_IMGFLAG_DIRECT;
    mpi->stride[0] = mpi->stride[1] = mpi->stride[2] = 0;
    mpi->planes[0] = mpi->planes[1] = mpi->planes[2] = NULL;
    // hack to get around a check and to avoid a special-case in vd_ffmpeg.c
    mpi->planes[0] = (void *)rndr;
    mpi->num_planes = 1;
```

```
    mpi->priv = rndr;
    mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: get_image }\n");
    return VO_TRUE;
}

static int query_format(uint32_t format)
{
    int default_flags = VFCAP_CSP_SUPPORTED | VFCAP_CSP_SUPPORTED_BY_HW | VFCAP_HWSCALE_UP | VFCAP_HWSCALE_
DOWN | VFCAP_OSD | VFCAP_EOSD | VFCAP_EOSD_UNSCALED | VFCAP_FLIP;
    switch (format) {
    case IMGFMT_BGRA:
        if (force_mixer)
            return 0;
    case IMGFMT_YV12:
    case IMGFMT_I420:
    case IMGFMT_IYUV:
    case IMGFMT_NV12:
    case IMGFMT_YUY2:
    case IMGFMT_UYVY:
        return default_flags | VOCAP_NOSLICES;
    case IMGFMT_VDPAU_MPEG1:
    case IMGFMT_VDPAU_MPEG2:
    case IMGFMT_VDPAU_H264:
    case IMGFMT_VDPAU_WMV3:
    case IMGFMT_VDPAU_VC1:
    case IMGFMT_VDPAU_MPEG4:
        if (create_vdp_decoder(format, 48, 48, 2))
            return default_flags;
    }
    return 0;
}

static void DestroyVdpauObjects(void)
{
    int i;
    VdpStatus vdp_st;

    free_video_specific();

    for (i=0; i < 2; i++) {
      vdp_st = vdp_presentation_queue_destroy(VdpStream[i].vdp_flip_queue);
      CHECK_ST_WARNING("Error when calling vdp_presentation_queue_destroy")
      vdp_st = vdp_presentation_queue_target_destroy(VdpStream[i].vdp_flip_target);
      CHECK_ST_WARNING("Error when calling vdp_presentation_queue_target_destroy")
    }

    for (i = 0; i <= NUM_OUTPUT_SURFACES; i++) {
        vdp_st = vdp_output_surface_destroy(output_surfaces[i]);
        CHECK_ST_WARNING("Error when calling vdp_output_surface_destroy")
        output_surfaces[i] = VDP_INVALID_HANDLE;
    }

    for (i = 0; i<eosd_surface_count; i++) {
        if (eosd_surfaces[i].surface != VDP_INVALID_HANDLE) {
            vdp_st = vdp_bitmap_surface_destroy(eosd_surfaces[i].surface);
            CHECK_ST_WARNING("Error when calling vdp_bitmap_surface_destroy")
        }
        eosd_surfaces[i].surface = VDP_INVALID_HANDLE;
    }

    vdp_st = vdp_device_destroy(vdp_device);
    CHECK_ST_WARNING("Error when calling vdp_device_destroy")
}

static void uninit(void)
{
    int i;

    if (!vo_config_count)
        return;
    visible_buf = 0;

    for (i = 0; i < MAX_VIDEO_SURFACES; i++) {
        // Allocated in ff_vdpau_add_data_chunk()
        av_freep(&surface_render[i].bitstream_buffers);
        surface_render[i].bitstream_buffers_allocated = 0;
    }

    // Destroy all vdpau objects
    DestroyVdpauObjects();

    free(index_data);
    index_data = NULL;

    free(eosd_surfaces);
    eosd_surfaces = NULL;
    free(eosd_targets);
    eosd_targets = NULL;

#ifdef CONFIG_XF86VM
    vo_vm_close();
#endif
    vo_x11s_uninit();
}

static const opt_t subopts[] = {
    {"deint",    OPT_ARG_INT,   &deint,    (opt_test_f)int_non_neg},
    {"chroma-deint", OPT_ARG_BOOL,  &chroma_deint,  NULL},
    {"pullup",   OPT_ARG_BOOL,  &pullup,  NULL},
    {"denoise", OPT_ARG_FLOAT, &denoise, NULL},
    {"sharpen", OPT_ARG_FLOAT, &sharpen, NULL},
    {"colorspace", OPT_ARG_INT, &colorspace, NULL},
    {"force-mixer", OPT_ARG_BOOL, &force_mixer, NULL},
    {"hqscaling", OPT_ARG_INT, &hqscaling, (opt_test_f)int_non_neg},
    {NULL}
};

static const char help_msg[] =
    "\n-vo vdpau command line help:\n"
    "Example: mplayer -vo vdpau:deint=2\n"
    "\nOptions:\n"
    "  deint (all modes > 0 respect -field-dominance)\n"
    "    0: no deinterlacing\n"
    "    1: only show first field\n"
    "    2: bob deinterlacing\n"
    "    3: temporal deinterlacing (resource-hungry)\n"
    "    4: temporal-spatial deinterlacing (very resource-hungry)\n"
    "  chroma-deint\n"
    "    Operate on luma and chroma when using temporal deinterlacing (default)\n"
    "    Use nochroma-deint to speed up temporal deinterlacing\n"
    "  pullup\n"
    "    Try to apply inverse-telecine (needs temporal deinterlacing)\n"
    "  denoise\n"
    "    Apply denoising, argument is strength from 0.0 to 1.0\n"
    "  sharpen\n"
    "    Apply sharpening or softening, argument is strength from -1.0 to 1.0\n"
    "  colorspace\n"
    "    0: guess based on video resolution\n"
    "    1: ITU-R BT.601 (default)\n"
    "    2: ITU-R BT.709\n"
    "    3: SMPTE-240M\n"
    "  hqscaling\n"
    "    0: default VDPAU scaler\n"
    "    1-9: high quality VDPAU scaler (needs capable hardware)\n"
    "  force-mixer\n"
    "    Use the VDPAU mixer (default)\n"
    "    Use noforce-mixer to allow BGRA output (disables all above options)\n"
    ;

static int preinit(const char *arg)
{
    int i;

  mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: preinit {\n");

    deint = 0;
    deint_type = 3;
    deint_counter = 0;
    deint_buffer_past_frames = 0;
    deint_mpi[0] = deint_mpi[1] = NULL;
    chroma_deint = 1;
    pullup  = 0;
    denoise = 0;
    sharpen = 0;
    colorspace = 1;
    force_mixer = 1;
    hqscaling = 0;
    if (subopt_parse(arg, subopts) != 0) {
        mp_msg(MSGT_VO, MSGL_FATAL, help_msg);
        return -1;
    }
    if (deint)
        deint_type = deint;
    if (deint > 1)
        deint_buffer_past_frames = 1;
    if (colorspace < 0 || colorspace > 3) {
        mp_msg(MSGT_VO, MSGL_WARN, "[vdpaustereo] Invalid color space specified, "
                "using BT.601\n");
        colorspace = 1;
    }

    if (!vo_s_init() || win_x11s_init_vdpau_procs())
        return -1;

    decoder = VDP_INVALID_HANDLE;
    for (i = 0; i < MAX_VIDEO_SURFACES; i++)
        surface_render[i].surface = VDP_INVALID_HANDLE;
    video_mixer = VDP_INVALID_HANDLE;
    for (i = 0; i <= NUM_OUTPUT_SURFACES; i++)
        output_surfaces[i] = VDP_INVALID_HANDLE;

    for (i = 0; i < 2; i++)
        VdpStream[i].vdp_flip_queue = VDP_INVALID_HANDLE;

    output_surface_width = output_surface_height = -1;

    // full grayscale palette.
    for (i = 0; i < PALETTE_SIZE; ++i)
        palette[i] = (i << 16) | (i << 8) | i;
    index_data = NULL;
    index_data_size = 0;

    eosd_surface_count = eosd_render_count = 0;
    eosd_surfaces = NULL;
    eosd_targets  = NULL;

    procamp.struct_version = VDP_PROCAMP_VERSION;
    procamp.brightness = 0.0;
    procamp.contrast   = 1.0;
    procamp.saturation = 1.0;
```

```
    procamp.hue        = 0.0;

  mp_msg(MSGT_VO, MSGL_DBG2, "[vdpaustereo] MOLDEO: preinit }\n");

    return 0;
}

static int get_equalizer(char *name, int *value)
{
    if (!strcasecmp(name, "brightness"))
        *value = procamp.brightness * 100;
    else if (!strcasecmp(name, "contrast"))
        *value = (procamp.contrast-1.0) * 100;
    else if (!strcasecmp(name, "saturation"))
        *value = (procamp.saturation-1.0) * 100;
    else if (!strcasecmp(name, "hue"))
        *value = procamp.hue * 100 / M_PI;
    else
        return VO_NOTIMPL;
    return VO_TRUE;
}

static int set_equalizer(char *name, int value)
{
    if (!strcasecmp(name, "brightness"))
        procamp.brightness = value / 100.0;
    else if (!strcasecmp(name, "contrast"))
        procamp.contrast = value / 100.0 + 1.0;
    else if (!strcasecmp(name, "saturation"))
        procamp.saturation = value / 100.0 + 1.0;
    else if (!strcasecmp(name, "hue"))
        procamp.hue = value / 100.0 * M_PI;
    else
        return VO_NOTIMPL;

    return update_csc_matrix();
}

static int control(uint32_t request, void *data, ...)
{
    if (handle_preemption() < 0)
        return VO_FALSE;

    switch (request) {
    case VOCTRL_GET_DEINTERLACE:
        *(int*)data = deint;
        return VO_TRUE;
    case VOCTRL_SET_DEINTERLACE:
        if (image_format == IMGFMT_BGRA)
            return VO_NOTIMPL;
        deint = *(int*)data;
        if (deint)
            deint = deint_type;
        if (deint_type > 2) {
            VdpStatus vdp_st;
            VdpVideoMixerFeature features[1] =
                {deint_type == 3 ?
                 VDP_VIDEO_MIXER_FEATURE_DEINTERLACE_TEMPORAL :
                 VDP_VIDEO_MIXER_FEATURE_DEINTERLACE_TEMPORAL_SPATIAL};
            VdpBool feature_enables[1] = {deint ? VDP_TRUE : VDP_FALSE};
            vdp_st = vdp_video_mixer_set_feature_enables(video_mixer, 1,
                                                         features,
                                                         feature_enables);
            CHECK_ST_WARNING("Error changing deinterlacing settings")
            deint_buffer_past_frames = 1;
        }
        return VO_TRUE;
    case VOCTRL_PAUSE:
        return int_pause = 1;
    case VOCTRL_RESUME:
        return int_pause = 0;
    case VOCTRL_QUERY_FORMAT:
        return query_format(*(uint32_t *)data);
    case VOCTRL_GET_IMAGE:
        // ##
        return get_image(data);
    case VOCTRL_DRAW_IMAGE:
        // ##
        return draw_image(data);
    case VOCTRL_GUISUPPORT:
        return VO_TRUE;
    case VOCTRL_BORDER:
        vo_x11s_border();
        resize();
        return VO_TRUE;
    case VOCTRL_FULLSCREEN:
        vo_x11s_fullscreen();
        resize();
        return VO_TRUE;
    case VOCTRL_GET_PANSCAN:
        return VO_TRUE;
    case VOCTRL_SET_PANSCAN:
        resize();
        return VO_TRUE;
    case VOCTRL_SET_EQUALIZER: {
        va_list ap;
        int value;
        if (image_format == IMGFMT_BGRA)
            return VO_NOTIMPL;
        va_start(ap, data);
        value = va_arg(ap, int);

        va_end(ap);
        return set_equalizer(data, value);
    }
    case VOCTRL_GET_EQUALIZER: {
        va_list ap;
        int *value;

        va_start(ap, data);
        value = va_arg(ap, int *);

        va_end(ap);
        return get_equalizer(data, value);
    }
    case VOCTRL_ONTOP:
        vo_x11s_ontop();
        return VO_TRUE;
    case VOCTRL_UPDATE_SCREENINFO:
        update_xinerama_info_S(&ScreenLeft);
        return VO_TRUE;
    case VOCTRL_DRAW_EOSD:
        if (!data)
            return VO_FALSE;
        generate_eosd(data);
        draw_eosd();
        return VO_TRUE;
    case VOCTRL_GET_EOSD_RES: {
        mp_eosd_res_t *r = data;
        r->mt = r->mb = r->ml = r->mr = 0;
        if (vo_fs) {
            r->w = vo_screenwidth;
            r->h = vo_screenheight;
            r->ml = r->mr = border_x;
            r->mt = r->mb = border_y;
        } else {
            r->w = vo_dwidth;
            r->h = vo_dheight;
        }
        return VO_TRUE;
    }
    }
    return VO_NOTIMPL;
}

/* @} */
```