# *DeckLink device in the Blender Game Engine*

## 1. Introduction

BlackMagicDesign is the manufacturer of a range of fast video capture and playback cards. The name code for these cards is 'DeckLink'. There is a full range of cards, from low cost to very high end but they all support the same generic API. This API has been implemented in the Blender Game Engine to allow real-time video processing features in the BGE.

Both functions, capture and playback, are available from Python in the VideoTexture module. This document describes the VideoTexture API for DeckLink devices and gives some details on the implementation.

## 2. Installation

Before a DeckLink device can be used in the BGE, the hardware must be installed. It can be a PCIe card or a USB device. This development has been done with the 'DeckLink 4K Extreme' card but all DeckLink devices should be supported. The 'Desktop Video' software package version 10.4 or above must be installed on the host as described in the DeckLink documentation.

In case you want to use video capture in Windows and if you have a recent nVideo Quadro card, you can benefit from the 'GPUDirect' technology to push the captured video frame very efficiently to the GPU. For this you need to install the 'DeckLink SDK' version 10.4 or above and copy the 'dvp.dll' runtime library to Blender's installation directory or to any other place where Blender can load a DLL from. This file is found in
Win/Samples/LoopThroughWithOpenGLCompositing/NVIDIA_GPUDirect/bin/win32
or
Win/Samples/LoopThroughWithOpenGLCompositing/NVIDIA_GPUDirect/bin/x64
whether you are using a 32bit or 64bit Blender build.

## 3. Capture

Capture is done with the class bge.texture.VideoDeckLink

### 3.1 VideoDeckLink Constructor

```
bge.texture.VideDeckLink(format,capture=0)
```

**format**: string describing the video format to be captured. The format of the string is
"<displayMode>/<pixelFormat>[/3D]" where <displayMode> describes the frame size and rate and <pixelFormat> the encoding of the pixels. The optional /3D suffix is to be used if the video stream is stereo with a left and right eye feed.
The valid <displayMode> values are copied from the 'BMDDisplayMode' enum in the DeckLink API without the 'bmdMode' prefix. In case a mode that is not in this list is added in a later version of the SDK, it is also possible to specify the 4 letters of the internal code for that mode. You will find the internal code in the DeckLinkAPIModes.h file that is part of the SDK. Here is for reference the full list of supported display modes with their equivalent internal code:

```
      NTSC        'ntsc'
      NTSC2398    'nt23'
      PAL         'pal '
      NTSCp       'ntsp'
      PALp        'palp'
HD 1080 Modes
```

```
        HD1080p2398 '23ps'
        HD1080p24   '24ps'
        HD1080p25   'Hp25'
        HD1080p2997 'Hp29'
        HD1080p30   'Hp30'
        HD1080i50   'Hi50'
        HD1080i5994 'Hi59'
        HD1080i6000 'Hi60'
        HD1080p50   'Hp50'
        HD1080p5994 'Hp59'
        HD1080p6000 'Hp60'
```
HD 720 Modes
```
        HD720p50    'hp50'
        HD720p5994  'hp59'
        HD720p60    'hp60'
```
2k Modes
```
        2k2398      '2k23'
        2k24        '2k24'
        2k25        '2k25'
```
4k Modes
```
        4K2160p2398 '4k23'
        4K2160p24   '4k24'
        4K2160p25   '4k25'
        4K2160p2997 '4k29'
        4K2160p30   '4k30'
        4K2160p50   '4k50'
        4K2160p5994 '4k59'
        4K2160p60   '4k60'
```

Most of these names are self explanatory. If necessary refer to the DeckLink API documentation for more information.

Similarly, <pixelFormat> is copied from the BMDPixelFormat enum. Here is for reference the full list of supported pixel format and their equivalent internal code:

```
        8BitYUV     '2vuy'
        10BitYUV    'v210'
        8BitARGB    * no equivalent code *
        8BitBGRA    'BGRA'
        10BitRGB    'r210'
        12BitRGB    'R12B'
        12BitRGBLE  'R12L'
        10BitRGBXLE 'R10l'
        10BitRGBX   'R10b'
```

Refer to the DeckLink SDK documentation for a full description of these pixel format. It is important to understand them as the decoding of the pixels is NOT done in VideoTexture for performance reason. Instead a specific shader must be used to decode the pixel in the GPU. Two shaders, for the '10BitYUV' and '10BitRGB' pixel format, are provided for reference in annexe A of this document.

Example: “HD1080p24/10BitRGB/3D” is equivalent to “24ps/r210/3D” and represents a full HD stereo feed at 24 frame per second.

Although video format auto detection is possible with certain DeckLink devices, the corresponding API is NOT implemented in the BGE. Therefore it is important to specify the format string that matches exactly the video feed. If the format is wrong, no frame will be captured. It should be noted that the pixel format that you need to specify is not necessarily the actual format in the video feed. For example, the 4K Extreme card delivers 8bit RGBs pixels in the '10BitRGB' format. Experimenting with the 'Media Express' application included in 'Desktop Video' will help you discover which pixel format works for a particular feed.

**capture**:Index of the DeckLink board to be opened, 0=first.

The returned object is a valid video source for the bge.texture.Texture object. However, it cannot be used in conjunction with filters and mixers because in this implementation the image data is not available at Python level; it can only be used as direct source of a bge.texture.Texture object and the frame is sent unaltered to the GPU.
Note: This implies that the GPU supports non power-of-two texture size, which is certainly the case for all modern GPUs.

Only the ' 8BitARGB' and '8BitBGRA' pixel format have an equivalent in OpenGL. Other formats are sent to the GPU as a 'GL_RED_INTEGER' texture (i.e. a texture with only the red channel coded as an unsigned 32 bit integer). This helps extracting the color channels in a shader as demonstrated in the 'r210' and 'v210' shaders in annexe A.

This constructor will throw a runtime error if the DeckLink device cannot be opened or the DVP library cannot be initialized.

### DeckLink internals

This constructor starts by acquiring an instance of the IdeckLinkIterator object. This is done in Linux by calling CreateDeckLinkIteratorInstance and in Windows by calling the generic COM function CoCreateInstance. The iterator object is the entry point of all DeckLink API calls. The following functions are called in sequence:

IdeckLinkIterator::Next() : to get the $n^{th}$ device as indicated by the capture parameter.
IdeckLink::QueryInterface() : to check if the device supports input, it returns a IdeckLinkInput object.
IdeckLinkInput::GetDisplayModeIterator() : to get a display mode iterator for the device.
IdeckLinkDisplayModeIterator::Next() : scan the modes to find one matching the format parameter.
IdeckLinkDisplayMode::GetDisplayMode()
IdeckLinkDisplayMode::GetFlags()
IdeckLinkInput::DoesSupportVideoMode() : to check if the device supports the mode.
IdeckLinkInput::SetVideoInputFrameMemoryAllocator() : to use a custom memory allocator.
This is necessary to ensure that frame buffers are aligned on a page boundary as required by GPUDirect and the GL_AMD_pinned_memory extension. The memory allocator must implement the IdeckLinkMemoryAllocator interface, in particular the IdeckLinkMemoryAllocator::AllocateBuffer() and IdeckLinkMemoryAllocator::ReleaseBuffer() functions to respectively, allocate and free frame buffers. Allocation in Windows is done with VirtualAlloc() and in Linux with posix_memalign().
For best efficiency, the allocator implements a cache for frame buffers: when the frame is released, the ReleaseBuffer() is called on the buffer, but the allocator instead of freeing the memory saves it in a cache so that it can return it on the next AllocateBuffer (all frames use the same buffer size).
The management of the cache must be thread safe because the AllocateBuffer() and ReleaseBuffer() functions can be called from different threads during capture.
On creation, the allocator verifies if the GPU support GPUDirect (only on Windows) and if so calls the dvpInitGLContext() function. This function is implemented in the Blender intern module 'gpudirect': it loads the 'dvp.dll' runtime library and dynamically binds all the DVP functions to static memory pointers, it then calls dvpInitGLContext() from the library.
IdeckLinkInput::EnableVideoInput(): to active the device but not start the capture yet. This also triggers the allocation of 64 frame buffers. For full HD 3D frames, this represents as much as 1Gb of memory. There is no need to keep so many frames in memory, that's why the custom allocator returns NULL after 5 frames allocation, which will constitute the cache for the lifetime of the VideoDeckLink object.
IdeckLinkInput::SetCallback(): to register a IdeckLinkInputCallback object that will be called on the IdeckLinkInputCallback::VideoInputFrameArrived() function whenever a frame is available. Note that the function is called from a separate thread.

## 3.2 VideoDeckLink Attributes

### 3.2.1 status

Status of the capture.

Type: integer (1=ready to capture, 2=capturing, 3=stopped)

### 3.2.2 framerate

Capture frame rate as computed from the video format.
Type: float

### 3.2.3 valid

Tells if the **image** attribute can be used to retrieve the image.
Type: boolean. Always False in this implementation (the image is not available at python level)

### 3.2.4 image

The image buffer. Always None in this implementation.

### 3.2.5 size

The size of the frame in pixel.
Stereo frame have double the height of the video frame, i.e. 3D is delivered to the GPU as a single image in top-bottom order, left eye on top.
Type: 2-uple of integers.

### 3.2.6 scale, flip, filters

These attributes are inherited from the ImageBase object. They are not operational for this object and should not be used.

## 3.3 VideoDeckLink methods

### 3.3.1 play()

Kick-off the capture after creation of the object. It calls internally the IdeckLinkInput::FlushStreams() and IdeckLinkInput::StartStreams() functions.

### 3.3.2 pause()

Temporary stop the capture. Use play() to restart it. It calls internally the IdeckLinkInput::PauseStreams() function.

### 3.3.3 stop()

Stop the capture. It calls internally the IdeckLinkInput::StopStreams() function.

## 3.4 Processing of a frame

The following diagram shows the timing and the process of one frame from the moment it enters the device and the moment it appears on the monitor. It is written in a pseudo language that resembles Python but the implementation is C++. The function names for the DVP and DeckLink APIs are correct but the parameters are simplified. Refer to the in-line comments and relevant documentation for more details on the functions.

```
     Capture thread             BGE thread
=============================================================================
|t0: the frame start to enter the device
|
| # the interval between t0 and t1 varies with the DeckLink device
| # and the type of input. On HDMI input, the delay is typically in
| # the order of 2-3 frames (at 24fps, this is 84-125ms)
| # On SDI input, the delay should be just about 1 frame.
|
|t1: the frame is delivered to the application
|    VideoInputFrameArrived(newFrame):
|       LockCache()
|       # only one frame is kept pending, the goal of the implementation
|       # is to get the frames as quickly as possible to the BGE.
|       if cacheFrame:
|          cacheFrame.Release():
|       cacheFrame = newFrame
|       cacheFrame.AddRef()
|       UnlockCache()
|
|
|
|  # the time between t1 and t2 is variable because the BGE is not
|  # synchronized with the capture. It depends on the frame rate:
|  # at 60fps, the average delay will be around 10ms
|
|t2: The Python scripts calls refresh() on the Texture object. This triggers the
|    calcImage function on the VideoDeckLink object:
|                                VideoDeckLink.calcImage(texId):
|                                   LockCache()
|                                   frame = cacheFrame
|                                   cacheFrame = null
|                                   UnlockCache()
|                                   if frame:
|                                      frame.GetBytes(bytes)
|                                      allocator.transferFrame(bytes,texId):
|                                         # process the frame only if it is our memory
|                                         if not allocated[bytes]:
|                                            return
|                                         # create a DVP texture the first time
|                                         if not dvpTex:
|                                            dvpTex=dvpCreateGPUTextureGL(texId)
|                                         # transfer objects are cached as well
|                                         transfer = pinnedBuffers[bytes]
|                                         if not transfer:
|                                           # this pseudo code demonstrates the use of the
|                                           # DVP API. In Linux, standard OGL functions are
|                                           # used to transfer the frame
|                                           transfer=TextureTransferDvp(dvpTex,bytes):
|                                               dvpHdl = dvpCreateBuffer(bytes)
|                                               dvpBindToGLCtx(dvpHdl)
|                                           pinnedBuffers.add(bytes,transfer)
|                                         # perform the transfer
|                                         transfer.PerformTransfer():
|                                            # tells DVP that OGL no longer uses the texture
|                                            dvpMapBufferEndAPI(dvpTex)
|                                            dvpBegin()
|                                            # wait until all DVP operations are done
|                                            dvpMapBufferWaitDVP(dvpTex)
|                                            # initiate the memory copy
|                                            dvpMemcpyLined(dvpHdl, dvpTex)
|                                            # tells DVP that there is no more DVP operations
|                                            dvpMapBufferEndDVP(dvpTex)
|                                            dvpEnd()
|                                            # wait until the texture is ready for use
|                                            dvpMapBufferWaitAPI(dvpTex)
|                                      frame.Release()
|
```

|t3: The texture is ready for use in the GPU. The delay between between t2 and t3
|    depends how fast DMA can transfer the frame to the GPU.
|
|t4: The BGE frame render starts.
|
|t5: The BGE frame render is done.
|
|t6: The frame starts to appear on the monitor.
|    If the full Python, transfer and graphic processing can be done in less than one
|    BGE frame, then the total time between t2 and t6 is equal to one BGE frame
|    (20ms at 50 fps, 42ms at 24fps), regardless of t3, t4 and t5.
|    This is because the BGE waits for the next display refresh before starting a
|    new frame. If the BGE cannot process everything in less than one frame, input
|    frames will be skipped. Tests shows that a modern CPU and GPU can easily
|    keep up with 50fps for both input and BGE frame rate.
|    Assuming there are no skipped frames, the total latency between t0 and t6 is
|    given by this formula:
|
|      latency = DeckLink_delay+async_delay+bge_frame_duration
|
|    Considering that async_delay is between 0 and bge_frame_duration. The minimum
|    bound of latency is DeckLink_delay+bge_frame_duration and the maximum bound is
|    DeckLink_delay+2*bge_frame_duration.
|
|    Example: for an input video at 24fps on HDMI and 50fps BGE, the latency is between
|             103ms and 165ms, assuming 2-3 frame delay on HDMI.
|
|

# 4. Playback

Certain DeckLink devices can be used to playback video: the host must send video frames regularly for immediate or scheduled playback. The video feed is outputted on HDMI or SDI interfaces.
This implementation only supports the immediate playback mode via the bge.texture.DeckLink object. This object is similar to the bge.texture.Texture: it has a source attribute that is assigned one of the source object in the bge.texture module. Refreshing the DeckLink object causes the image source to be computed and sent to the DeckLink device for immediate transmission on the output interfaces. Keying is supported: it allows to composite the frame with an input video feed.

## 4.1 Constructor

bge.texture.DeckLink(cardIdx=0,format="")

**cardIdx**: Number of the card to be used for output (0=first card). It should be noted that DeckLink devices are usually half duplex: they can either be used for capture or playback but not both.

**Format**: String representing the display mode of the output feed. The default value is reserved for auto detection but it is currently not supported (it will generate a runtime error) and thus the video format must be explicitly specified. If keying is the goal (see keying attributes), the format must match exactly the input video feed, otherwise it can be any format supported by the device (there will be a runtime error if not).
The format of the string is "<displayMode>[/3D]".
Refer to the capture constructor to get the list of acceptable <displayMode>. The optional "/3D" suffix is used to create a stereo 3D feed. In that case the 'right' attribute must also be set to specify the image source for the right eye.
Note: The pixel format is not specified here because it is always BGRA. The alpha channel is used in keying to mix the source with the input video feed, otherwise it is not used. If a conversion is needed to match the native video format, it is done inside the DeckLink driver or device.

### DeckLink internals

This constructor starts by acquiring an instance of the IdeckLinkIterator object (see VideoDeckLink constructor). The following functions are called in sequence:

IDeckLinkIterator::Next() : to get the $n^{th}$ device as indicated by the cardIdx parameter.
IDeckLink::QueryInterface() : to return IDeckLinkAttributes interface and check if keying is supported.
IDeckLink::QueryInterface() : to check if output is supported and return a IDeckLinkOutput object.
IDeckLink::QueryInterface() : in case keying is supported, return a IDeckLinkKeyer object.
IDeckLinkOutput::GetDisplayModeIterator() : to get a display mode iterator for the device.
IDeckLinkDisplayModeIterator::Next() : scan the modes to find one matching the format parameter.
IDeckLinkDisplayMode::GetDisplayMode()
IDeckLinkDisplayMode::GetFlags()
IDeckLinkOutput::DoesSupportVideoMode() : to check if the device supports the video mode, BGRA pixel format and 3D if specified.
IDeckLinkOutput::EnableVideoOutput(): to start the output video stream.
IDeckLinkOutput::CreateVideoFrame(): to create a BGRA buffer to store the output frame. This function is called twice if the 3D is enabled: one for the left and one for the right eye. The frame buffers are created with the flag bmdFrameFlagFlipVertical, which activates the hardware flip of the frame: the first bytes of the buffer will be at the bottom of the output video. This is also how OpenGL frame buffers are received, hence no software flip is necessary when outputting the BGE frame buffer. However, video file don't require flipping, but a software flip on the ImageFFmpeg object will be necessary to undo the hardware flip set here.

## 4.2 Attributes

### 4.2.1 source

This attribute must be set to one of the bge.texture objects that can be used as a texture source: ImageViewport (BGE frame buffer), ImageRender (custom render), ImageFFmpeg (image file), etc. The source image is placed at the top of the frame buffer (bottom left corner of the output video due to flipping). If the image size does not fit exactly the frame size, the **extend** attribute determines what to do.
For best performance, the source image should match exactly the size of the output frame.
A further optimization is achieved if the image source object is ImageViewport or ImageRender set for whole viewport, flip disabled and no filter: the GL frame buffer is copied directly to the image buffer and directly from there to the DeckLink card (hence no buffer to buffer copy inside VideoTexture).

### 4.2.2 right

If the video format is stereo 3D, this attribute should be set to an image source object that will produce the right eye images. If the goal is to render the BGE scene in 3D, it can be achieved with the following setup:

- Assign an ImageViewport object to the **source** attribute: it will put the view from the scene camera to the left eye of the output video.
- Add a camera into the scene that represents the right eye: parent it to the scene camera, put it 7cm away to simulate eye separation and make it slightly converge. The convergence point defines the focus: an object sitting at this point will be 'on screen' while any object sitting between this point and the cameras will pop out of the screen.
- Make sure the cameras has the same aspect ratio than the frame buffer: set the X and Y resolutions in the Dimensions panel of the Render properties identical to the window size (or display if the BGE is fullscreen).
- In Pyrthon, create an ImageRender object on that right eye camera and assign it to this attribute.

For correct stereo effect, you must also adjust both camera angles to match the apparent display size. For example, if the output screen is a W meters width TV viewed at D meters distance, then set the Focal and Size values in the Camera properties such that Focal/Size = D/W. For a 1m TV viewed at 3m distance, set Focal to 30 and size to 10.
Note that enabling stereo in the BGE is not needed and not recommended for this setup (it will consume CPU for nothing).

### 4.2.3 keying

Boolean to determine if keying is enable.
False (default): the output frame is sent unmodified on the output interface (in that case no input video is required). Internally it calls IdeckLinkKeyer::Disable() to turn keying off in the device.
True: the output frame is mixed with the input video, using the alpha channel to blend the two images and the combination is sent on the output interface. Internally it calls the IDeckLinkKeyer::Enable() and IDeckLinkKeyer::SetLevel() function to enable keying and set the initial level.

### 4.2.4 level

Integer, 0 to 255.
Sets the keying level: it is an global alpha value that multiplies the alpha channel of the output frame. Use 255 (the default) to keep the alpha channel unmodified, 0 to make the output frame totally transparent. If keying is enabled, it calls IdeckLinkKeyer::SetLevel() internally to change the keying level in the device.

### 4.2.5 extend

Boolean to determine how to map the image if the image source does not fit the output frame size.
False (the default): map the image pixel by pixel. If the image size is smaller than the frame size, extra space around the image is filled with 0-alpha black. If it is larger, the image is cropped to fit the frame size.

True: the image is scaled by the nearest neighbor algorithm to fit the frame size. The scaling is fast but poor quality. For best results, always adjust the image source to match the size of the output video.

## 4.3 Methods

### 4.3.1 close()

Close the DeckLink device and release all resources. After calling this method, the object cannot be reactivated, it must be destroyed and a new DeckLink object created from fresh to restart the output.

### 4.3.2 refresh(refresh[, ts])

This method must be called frequently to update the output frame in the DeckLink device.

**refresh**: Boolean. True if the source objects image buffer should be invalidated after being used to compute the output frame. This triggers the recomputing of the source image on next refresh, which is normally the desired effect. False if the image source buffer should stay valid and reused on next refresh.
Note that the DeckLink device stores the output frame and replays until a new frame is sent from the host. Thus, it is not necessary to refresh the DeckLink object if it is known that the image source has not changed.

**ts**: Float. The timestamp value passed to the image source object to compute the image. If unspecified, the BGE clock is used.

### DeckLink internals

The image buffer for left (and right) eye is obtained from the source(s). If the size doesn't match exactly the output frame, the image is mapped, according to **extend**, to the frame buffer allocated in the constructor. The resulting image, or original image if no mapping was necessary, is sent to the device with IdeckLinkOutput::DisplayVideoFrameSync(). The frame is inserted in the output stream as quickly as possible, on the next video frame at best. This time is variable as the BGE is not synchronized with the device.

# A. Pixel shaders

This annexe lists shaders that can be used with the textures generated by the VideoDeckLink object.

## A.1 'r210' shader

The r210 pixel format is a 4:4:4 raw format with the RGB channels coded in 10 bits fields and packed in one 32 bits integer. This shader assumes that the color values in the 10 bits fields are coded with a reduced range such that null intensity is 64 and full intensity is 940.
Note: this corresponds the RGBs color space where the colors are coded over 8 bits in the reduced range 16-235.
If it is observed that a particular video feed produces full range colors (0->1023) instead of the reduce range, the shader must be adjusted accordingly.

```
#version 130
// use usampler2D as the pixel format is unsigned integer in VideoDeckLink
uniform usampler2D tex;

void main(void)
{
   vec4 color;
   unsigned int r;
   r = texture(tex, vec2(gl_TexCoord[0])).r;
   // reduced range 64->940, mind the 64U and 1/880 numeric values
   color.b = float((((r >> 24U)&0xFFU)+((r>>8U)&0x300U))-64U)*0.001136364;
   color.g = float((((r >> 18U)&0x3FU)+((r>>2U)&0x3C0U))-64U)*0.001136364;
   color.r = float((((r >> 12U)&0x0FU)+((r<<4U)&0x3F0U))-64U)*0.001136364;
   // Set alpha to 0.7 for partial transparency when GL_BLEND is enabled
   color.a = 0.7;
   gl_FragColor = color;
}
```

### A.2 'v210' shader

This pixel format is a 4:2:2 representation where twelve 10 bits chrominance and luminance channels representing six pixels are packed into four 32 bits integer. The channels are also using the reduced range 64->940.

```
#version 130
uniform usampler2D tex;
void main(void)
{
   vec4 color;
   float tx, ty;
   float width, sx, dx, Y, U, V, bx;
   unsigned int w0, w1, w2, w3;
   int px;

   tx = gl_TexCoord[0].x;
   ty = gl_TexCoord[0].y;
   // size of the texture (=real size * 2/3)
   width = float(textureSize(tex, 0).x);
   // to sample macro pixels (6 pixels in 4 words)
   sx = tx*width*0.25+0.05;
   // index of display pixel in the macro pixel 0..5
   px = int(floor(fract(sx)*6.0));
   // increment as we sample the macro pixel
   dx = 1.0/width;
   // base x coord of macro pixel
   bx = (floor(sx)+0.01)*dx*4.0;
   // load the 4 words components of the macro pixel
   w0 = texture(tex, vec2(bx, ty)).r;
   w1 = texture(tex, vec2(bx+dx, ty)).r;
   w2 = texture(tex, vec2(bx+dx*2.0, ty)).r;
   w3 = texture(tex, vec2(bx+dx*3.0, ty)).r;
   switch (px) {
   case 0:
   case 1:
      U = float( w0      &0x3FFU);
      V = float((w0>>20U)&0x3FFU);
      break;
   case 2:
   case 3:
      U = float((w1>>10U)&0x3FFU);
      V = float( w2      &0x3FFU);
      break;
   default:
      U = float((w2>>20U)&0x3FFU);
      V = float((w3>>10U)&0x3FFU);
      break;
   }
   switch (px) {
   case 0:
      Y = float((w0>>10U)&0x3FFU);
      break;
   case 1:
      Y = float( w1      &0x3FFU);
      break;
   case 2:
      Y = float((w1>>20U)&0x3FFU);
      break;
   case 3:
```

```
      Y = float((w2>>10U)&0x3FFU);
      break;
   case 4:
      Y = float( w3      &0x3FFU);
      break;
   default:
      Y = float((w3>>20U)&0x3FFU);
      break;
   }
   // reduced range 64->940
   Y = (Y-64.0)*0.001141553;
   U = (U-64.0)*0.001116071-0.5;
   V = (V-64.0)*0.001116071-0.5;
   color.r = Y + 1.5748 * V;
   color.g = Y - 0.1873 * U - 0.4681 * V;
   color.b = Y + 1.8556 * U;
   color.a = 0.7;
   gl_FragColor = color;
}
```