Dolby Vision

Content Editing for Third-Party Developers

15 May 2025
Version 2.0

Notices

Trademarks

Patents

Contents

1

Introduction

This documentation focuses on video editing of Dolby Vision files in an Android application, and covers support provided in a sample application for critical aspects on both graphics processing unit (GPU) and display processing unit (DPU) platforms and corresponding Dolby Vision versions.

Overview of documentation

This documentation focuses on video editing and previewing of Dolby Vision files, which may involve any of the following operations:

- Display management
- Editing, including applying pixel-level effects and overlaying text
- Hybrid log-gamma signaling and previewing
- Trimming and remultiplexing

This documentation has a chapter for each of these operations.

Your application may support some or all of these operations for standard dynamic range (SDR). However, to support Dolby Vision, the procedures used to carry out these operations may differ from your current implementation.

Overview of Dolby Vision

Dolby Vision empowers visual creatives to add ultravivid colors, sharper contrast, and richer details to their visual creations.

It unlocks the full potential of high-dynamic-range imaging (HDR) technology by dynamically optimizing image quality based on your device, platform, and service.

Dolby enables capturing, editing, viewing, and sharing videos in Dolby Vision on a variety of Android Smartphone devices.

Dolby welcomes creative application developers and content providers to take advantage of the new capabilities of these devices, update their experiences, and allow their users to make and to enjoy Dolby Vision content with exceptional qualities not found in standard video.

Dolby Vision capture versions and platforms

There are two different versions of Dolby Vision capture, for different display management processing units or platforms:

- A version using the graphics processing unit (GPU) of the mobile phone
- A version using the display processing unit (DPU) of the Qualcomm Snapdragon 8 Generation 2 chip

Determining the version and platform

Some devices have a DPU, but still implement display management processing with their GPU. For this case, the sample application provides the `getSupportStatus` function, located in `app/src/main/java/com/dolby/capture/filtersimulation/CodecBuilderImpl.java`, which detects the version of Dolby Vision capture installed on the device. The version installed corresponds to the display management processing unit or platform. A function like this should always be used to determine processing appropriate to the display management hardware implementation.

Applying a lookup table for the version and platform

For Dolby Vision playback, a 3D lookup table must be applied to every frame.

For a version and platform using a GPU, this requires an extra step. The sample code includes a 3D lookup table that is a best fit for the majority of DCI-P3-compliant displays. The sample application also shows how to apply this lookup table to a texture.
For a version and platform using the Qualcomm DPU, the DPU acts as a hardware lookup table. The Qualcomm DPU provides much faster and more power-efficient lookup table application than a GPU. To enable this, frames sent to the display must be labeled as HLG (hybrid log-gamma). For applications using OpenGL, this is not easy. There is no way to define a HLG context in OpenGL. Android, however, provides the `ImageWriter` class to label surfaces as HLG. (See the information regarding preview, transcoding, and hybrid log-gamma signaling.)

Differences with Dolby Vision

Color space

Editing supports either one of the ITU-R Recommendation BT.709 (BT.709) and ITU-R Recommendation BT.2020 (BT.2020) color standards. The correct color standard is signaled by the input decoder and the corresponding color space conversions are selected automatically in the editing GPU shader.

Encoder

Phones that support Dolby Vision have an encoder specific to Dolby Vision. This encoder expects BT.2020 limited range YUV HLG input.

Decoder

Phones that support Dolby Vision have a decoder specific to Dolby Vision. This decoder communicates the output colour space to the editor application, which influences the behavior of the shader system.

Multiplexing

Dolby Vision content requires proper multiplexing to ensure optimal playback and compatibility. After operations such as trimming or transcoding, remultiplexing is essential to maintain the integrity of the Dolby Vision signaling within the container.

Sample application

To assist in your integration, a sample application is provided that integrates Dolby Vision video editing. Reviewing the sample application and other materials provided with this documentation is critical to ensuring that your integration is successful.

Although the Android application programming interface (API) calls used in the sample application are in many cases nonstandard, they are backward compatible with all standard video formats.

This documentation highlights important differences and explains why these differences are necessary. The more critical parts of sample application code are included and explained here. For more details and context, please refer directly to the sample application code and its comments.

Setting up the sample application

Download the Dolby sample application for video editing from GitHub at https://github.com/DolbyLaboratories/dolby-vision-editor.

Download Android Studio at https://developer.android.com/studio.

After installing Android Studio, open the sample application. Android Studio recognizes the sample application as an Android project, then downloads libraries and builds the project.

Note:

This documentation is applicable only for applications that require Android 13 as a minimum. Before proceeding, please ensure that your code is compliant with Android 13 and does not rely on any deprecated functions.

The sample application does not support combining video files.

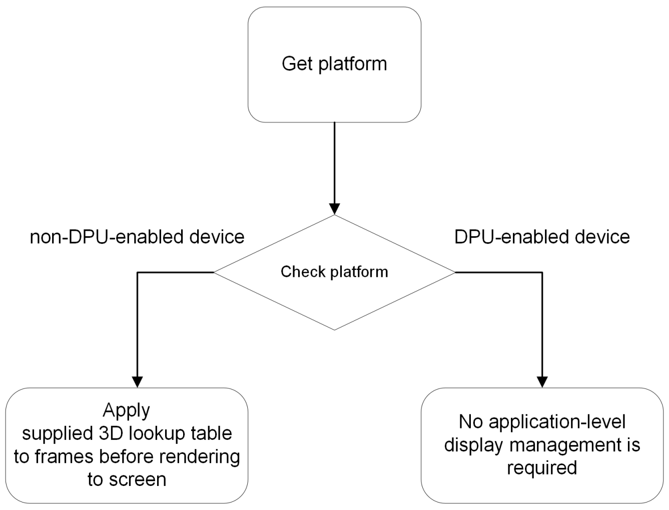
The sample application has been tested only with Dolby Vision profile 8.4. No other profiles are supported. See What are Dolby Vision profiles? for more information.

2

Display management

There are two versions of Dolby Vision capture; the version present on a device depends on the display management processing unit or platform used for Dolby Vision. It is critical that an application can determine which version is present, and can run correctly on both versions. Determining the version reliably involves examining the Dolby Vision codecs present on the device.

The following diagram shows an example of application logic that depends on the version of Dolby Vision capture.


Get platform
non-DPU-enabled device
Check platform
DPU-enabled device
Apply supplied 3D lookup table to frames before rendering to screen
No application-level display management is required

The difficulty in determining the version of Dolby Vision capture present on a device is that not much of the information available on the device can be used reliably. The presence of a DPU system-on-chip (SoC) is not sufficient to determine the Dolby Vision capture version, because there are devices running GPU versions on hardware with a DPU. The most reliable way of determining the version is to examine the Dolby Vision codecs present on the device. Different names are given to the two versions in their codecs.

It is sufficient to examine only one component (either the decoder or the encoder) for its name. The following function examines the decoder. The names of the decoder component in the non-DPU-enabled version are:

```
c2.dolby.decoder.hevc
c2.dolby.decoder.hevc.secure
c2.dolby.avc-hevc.decoder
c2.dolby.avc-hevc.decoder.secure
c2.mtk.hevc.decoder
c2.mtk.hevc.decoder.secure
```

The names of the decoder in the DPU-enabled version are:

```
c2.qti.dv.decoder
c2.qti.dv.decoder.secure
```

The names of the components are similar. For the encoder names, the term 'decoder' is replaced by 'encoder'.

The sample application uses the `getSupportStatus` function to determine the version based on the name of the decoder component. The `getSupportStatus` function is defined in `app/src/main/java/com/dolby/capture/filtersimulation/CodecBuilderImpl.java`.

The `getSupportStatus` function first gets a list of codec information about every codec on the device, and then filters out codec information without a GPU or DPU decoder name. Finally, from the remaining decoder names, it collects indicators of which platform is supported and returns a platform indicator.

```
public static PlatformSupport getSupportStatus()
{
    MediaCodecList mediaCodecList = new
MediaCodecList(MediaCodecList.REGULAR_CODECS);

    ArrayList<PlatformSupport> supports =
Arrays.stream(mediaCodecList.getCodecInfos()).filter(x -> {
       return x.getCanonicalName().equals(DOLBY_GPU_DECODER)
               || x.getCanonicalName().equals(DOLBY_GPU_DECODER_MTK_VPP)
               || x.getCanonicalName().equals(DOLBY_DPU_DECODER);
    }).map(x ->
    {
      switch (x.getCanonicalName())
      {
          case DOLBY_GPU_DECODER:
          case DOLBY_GPU_DECODER_MTK_VPP:
              return PlatformSupport.GPU;
          case DOLBY_DPU_DECODER:
              return PlatformSupport.DPU;
          default:
              return PlatformSupport.NONE;
      }
    }).collect(Collectors.toCollection(ArrayList::new));

    if(supports.isEmpty())
    {
        return PlatformSupport.NONE;
    }
    else
    {
        return supports.get(0);
    }
}
```

Editing using the GPU shaders

The sample application provides two GPU (graphics processing unit) shaders for editing. The DPU (display processing unit) is not used for editing, but is used for display.

Editing supports either BT.709 or BT.2020 color standards. The correct color standard is signaled by the input decoder. The corresponding color space conversions are selected automatically in the editing GPU shader.

Using these GPU shaders involves:

- Choosing a shader
- Selecting a processing approach (single-pixel or quad-pixel)
- Using uniform blocks and parameters
- Adjusting image texture dimensions
- Managing resources for shader compilers
- Timing GPU execution
- Debugging

Locating the shaders

The two GPU shaders and their editing functions are in subfolders of `editor\app\src\main`.

C++ code for the shaders is in the `cpp` subfolder.

Java code for an input surface and a filter simulation is in the `java/com/dolby/capture` subfolder.

The compute and copy shaders

The sample code includes two shaders as basic components:

- A compute shader that implements all editing functionality
- A simple vertex and fragment copy shader that renders output in a manner compatible with Android rendering

The compute shader provides efficiencies that offset the incremental costs of using the copy shader, and performs better than a single vertex and fragment copy shader that implements all operations.

Using the compute shader and its processing approaches

The compute shader has two GPU editing functions, defined in `EditShadersText.cpp`, which implement two approaches to processing:

- The `editPixel` function in `EditShaders::mEditLibrary` implements the single-pixel approach, a conventional programming of the GPU for processing a single pixel at a time.
- The `editPixel` function in `EditShaders::mEditLibraryQuad` implements the quad-pixel approach, which processes four pixels at a time for better performance and lower power consumption.

The compute shader engine allows a bitmap image to be composited into the video stream. The bitmap image must meet the following requirements for blending:

- The bitmap image must be a four channel RGBA image.

  In the sample code, the A channel is used as an alpha key to control the blending of the bitmap with the video.
- The bitmap image must have the same dimensions as the video stream.

There are two means of configuring and controlling the compute shader:

- By `#define` directives (when the shader text is assembled and compiled)
- By run-time controls passed in a uniform block

The single-pixel approach

The compute shader edit function for the single-pixel approach processes one pixel at a time.

The single-pixel approach is included primarily for illustration of editing functions, but could be used in the copy shader (if needed for some application).

The single-pixel edit function is declared as follows, with its implementation in `EditShadersText.cpp`:

```
vec4 editPixel( vec4 originalPixel, vec4 compositePixel, vec2 vidCoords, vec2
fCoords)
```

In the shader code, if the input is YUV, it is converted to RGB by the `icsc` function. If the editing is to be performed in BT.2020, input conversion is performed by the `sRGB2L` function and compositing texture by the `L2HLG` function, to upconvert to HLG.

Edit functions are performed using both YUV and RGB. The `editYUV` and `editRGB` functions are used to convert pixels back and forth in the editing domain. The operations in these functions vary depending on whether the editing is performed in BT.709 or BT.2020. Gain and offset are applied in RGB; contrast and saturation are applied in YUV. A rudimentary gamut limitation is performed before YUV override values are then applied.

Switching back to RGB, the text or graphics compositing texture is then blended with the input using the A channel in the compositing image.

Four wipers between original and processed values are applied, although only the left wiper is exposed in the Java interface. A zebra pattern is then applied to mark both range and gamut excursions.

Finally, the output color-space conversion is performed, using a static look-up table (LUT) (see `hlg_lut_500_p3_33.h` and `hlg_lut_500_p3_33.cpp`).

Color space changes are performed conventionally, using 3×3 matrices and matrix and vector multiplication. Because OpenGL is column-major rather than row-major, the matrices are transposed as compared to C++.

The color-space conversion function used with the single-pixel approach is declared as follows with its implementation in `EditShadersText.cpp`:

```
vec4 ColorSpaceConversion(mat3 csc, vec4 pixel)
```

The quad-pixel approach

The compute shader edit function for the quad-pixel approach processes four pixels at a time.

The quad-pixel edit function is declared as follows with its implementation in `EditShadersText.cpp`:

```
mat4 editPixel( mat4 originalPixel, mat4 compositePixel, mat2x4 vidCoords, mat4x2
fCoords)
```

A `mat4` matrix is used as a convenient container to hold four pixels. This makes it easy to pass four pixels as inputs to and outputs from functions in the shader.

OpenGL is column-major, meaning that color components are adjacent when sorted by component, producing convenient `vec4` vectors. For example, `mtx[0]` is a `vec4` of the red (or luma) components for the four pixels. Processing by color components permits better utilization of the four vectorized computation channels in the GPU.

Four pixels are read at a time into the `mat4` matrix, effectively sorted by pixel. The matrix is then transposed to sort by component for processing. Finally, the matrix is transposed back to sort by pixel to write out the pixels.

Color-space changes still use the same 3×3 matrices, but instead process four pixels at a time, sorted by component.

The color-space conversion function used with the quad-pixel approach is declared as follows with its implementation in `EditShadersText.cpp`:

```
mat4 ColorSpaceConversion(mat3 csc, mat4 pixels)
```

The computation can be accomplished using just six fused multiply-and-add operations and three multiply instructions.

Other subordinate functions are modified accordingly, although some still process pixels individually where an efficient quad-pixel approach may not exist.

For both the initial RGB and YUV offset, gain, contrast and saturation calculations, four pixels can be computed using the same number of instructions as processing three of them individually.

The `Gamut` function illustrates more complicated processing of four pixels at a time, sorted by component. The `Zebra` function illustrates how to apply logical conditions over four pixels at a time without any branching.

The forced YUV override functions take the same number of instructions to process four pixels as it takes to process one.

Conventional 4×4 homogenous transform matrices can be applied to the input and compositor images, if desired. Full 3D digital video effects engines can drive these transform matrices.

There are two compute shader controls for selecting the quad-pixel approach instead of the single-pixel approach:

```
// Compute shader configuration options
int mComputeShaderQuadFactor    = 4;
eShaderEditMode mShaderEditMode = eShaderEditModeFourPixels;
```

Selecting a processing approach

The default approach of the compute shader is the quad-pixel approach.

Setting the shader edit mode as follows instead selects the single-pixel approach:

```
eShaderEditMode mShaderEditMode = eShaderEditModeSinglePixel;
```

Changing the pixel factor from 4 to 1 as follows also selects the single-pixel approach:

```
int mComputeShaderQuadFactor = 1;
```

The `mComputeShaderQuadFactor` pixel factor configures each GPU core to read and to write either one or four pixels. When `mComputeShaderQuadFactor=1`, only the single-pixel edit function can be used. However, when `mComputeShaderQuadFactor=4`, either function may be used; the single-pixel edit function can be called four times, or the four-pixel edit function can be called once.

Because the quad-pixel approach processes four pixels at a time, the call to the `glDispatchCompute` function must be modified accordingly, multiplying by the pixel factor `mComputeShaderQuadFactor` as in `EditShaders.cpp`:

```
glDispatchCompute( DivUp(Width(), mWorkGroupSizeX * mComputeShaderQuadFactor),
                   DivUp(Height(), mWorkGroupSizeY),
                   1);
```

Using the copy shader

The vertex and fragment copy shader is implemented by methods of the `Renderer` class, which inherits from `EditShaders` and is defined in [`Renderer.h`] and [`Renderer.cpp`].

The copy shader uses a frame buffer object to communicate necessary control information to the smartphone screen in playback mode, and to the encoder in transcode mode. This frame buffer object communication is the primary reason for having the copy shader.

Using uniform blocks and parameters with shaders

The block class `EditShadersUniformBlock` is defined in [`EditShaders.h`]. The content of a block of this class must exactly match the declaration of the uniform `EditShadersUniformBlock` defined in [`EditShadersText.cpp`] in both size and packing.

Note that integers are passed as floats, and in accordance with the relevant Institute of Electrical and Electronics Engineers (IEEE) standard (IEEE 754), integers are represented exactly within the range of the mantissa.

Also, the `EffectParameters` `enum` declaration must exactly match the order of the parameters in the uniform block. This allows the uniform block data to be treated as a float array and addressed by enumerated value. By using the same `enum` declaration, code written in Java can also access these parameters efficiently by index.

Adjusting image texture dimensions

In some cases, the input image texture dimensions may be larger than the active area of the image itself. This can happen when the original input video was encoded with dimensions that were not evenly divisible by the macro block size of the encoder. Because the compute shader uses an input texture sampler that scales the unit interval across the input texture dimensions, not the active image area, an adjustment is made to the sampler input.

The input texture dimensions can be accessed by instance methods of the `Simulation::HardwareBuffer` class, and passed in the call to the renderer `RunEditComputeShader` method:

```
JNI_GLOBAL::renderer->RunEditComputeShader(inputImage.getBufferTexture(),
                                           inputImage.getHardwareBufferWidth(),
                                           inputImage.getHardwareBufferHeight());
```

These dimensions are used to compute adjustment ratios that are passed in the compute shader uniform block.

Managing resources for shader compilers

One of the more difficult details of handling multiple error exits from a shader compiler is releasing the intermediate OpenGL resources that may otherwise accumulate. For this, a lambda function is passed to the constructor of a `ScopeExitFunction` declared and implemented in [`Tools.h`]. The following code from [`Tools.cpp`] is an example of its usage:

```
// Used by the cleanup lambda
bool   guilty      = true; // guilty until proven innocent
GLuint main_shader = GL_INVALID_VALUE, vertex_shader = GL_INVALID_VALUE;

ScopeExitFunction cleanup([&]()
{
    DeleteShader(main_shader);

    DeleteShader(vertex_shader);

    if (guilty) DeleteProgram(program);

});
```

The `cleanup` lambda function is called at any point that the shader goes out of scope, and the lambda capture closure allows the `cleanup` function to access the local handles for the OpenGL resources that need to be released. This approach works even for uncaught exceptions when destructors are called as the stack is unwound.

Timing GPU execution

The class `ScopeTimerGPU` defined in `Tools.h` is used for performance measurement of GPU operations on a platform.

Debugging shader compiler errors

The `CompileAndLinkShader` function defined in `Tools.h` is useful for debugging.

The function quits on any error and outputs the most diagnostic information possible, especially for compiler errors. By default, in debug mode, source code for all shaders is output with line numbers, which is useful for shader source code that is spliced together at run time.

The error generated by OpenGL usually includes a line number. When working in Android Studio, double-click the line number in Logcat, then type **Crtl-F**, and an up arrow navigates to the erroneous line.

HLG signaling and previewing

Adding support for Dolby Vision editing involves understanding how to process hybrid log-gamma (HLG) frames correctly so that the preview surface is aware of their format.

Overview of hybrid log-gamma signaling

Hybrid log-gamma (HLG) image signaling is potentially one of the greatest obstacles in adding support for Dolby Vision editing to an existing application. Rather than making additions to existing code, this involves what could be a major alteration to the implementation of a pipeline.

To ensure correct operation for preview, the processing framework presented here must be used. The same processing framework can be used for transcoding, but this is not required for correct operation. In the sample application, the same processing framework is used for both preview and transcoding to make the application easier to manage.

Regardless of whether frames are sent to the screen to be previewed or to an encoder during transcoding or editing, the same processing framework applies:

- The video decoder produces frames onto a `Surface`.
- The surface is used as input to a set of OpenGL shaders that apply effects.
- The output from the shaders is rendered onto another surface, which could be either a screen or a video encoder.

The traditional approach to applying effects to a video frame in Android is to use `SurfaceTexture`, which allows use of the `MediaCodec` decoder instance to output a texture directly. A vertex and fragment shader can then be used to render the modified frame directly onto the screen or the encoder input surface. The issue with this approach is that OpenGL is the interface between the decoder and the screen, but is incapable of communicating to the screen that the pixels are in HLG, which is necessary to take advantage of DPU hardware.

The solution in the sample application decouples the decoding procedure from subsequent procedures as much as possible. The `MediaCodec` callback class has restrictions. The methods for input and output buffer handling must not perform thread blocking or waiting. This means not placing any kind of flow control in these calls. The decoder has access to a limited pool of buffers, and once these are filled, the decoder will stop processing.

The decoder implements an isolated producer and consumer pattern. The decoder callbacks are assigned to a single thread. When the decoder produces a buffer, it is placed into a queue. In order to avoid any blocking, the decoder uses an instance of the Java utility class `ConcurrentLinkedQueue`, which can guarantee thread safety without need for locks or semaphores. Adding the buffer to the queue is the only operation performed on output buffers in the decoder or the producer thread.

The video decoder class file `VideoDecoder.java` also defines a small inner static class `BufferProcessor`, which handles the consumer part of the pattern. This class starts another thread that is independent of the decoder so that the speed at which frames go to the screen can be throttled according to the frame rate of the input video. This is where some of the key changes to support Dolby Vision start to occur. The render target of the video decoder is not the screen or the encoder directly. Instead, the output is piped to an `ImageReader`.

The consumer thread loops to poll the queue for buffers. If the queue is not empty, it removes the lead element and asks the `MediaCodec` instance to render the buffer to the image reader. The callback that is triggered upon the image reader receiving a frame operates in a different thread than the decoder or the buffer consumer. This thread is set up as an OpenGL context, which is important for the next step.

Because the decoder output is decorated with the image reader, the output is seen as a series of image objects. The next step involves getting an OpenGL texture that can be used as input to the shader system. There is no API that can directly map an `Image` to an OpenGL resource. However, the image can be converted to a `HardwareBuffer`, which can be mapped to a texture. The sample application does this *via* Java Native Interface (JNI) (the Java Native Interface), the calls for which are made in the `FrameHandler` class. The next issue is that the output image from the decoder is read only, so an empty buffer is required to draw the result of the edits.

Obtaining an empty image that can be used for this purpose involves the `ImageWriter`. This utility is extremely useful. It allows for dataspace assignment that fixes the HLG issue, and it can render directly to any resource that uses a `Surface` object to accept input. Once an empty image is obtained from the writer, it can be mapped to a texture in the same way as the input image. As for the texture generation process, the following code from `editor/app/src/main/cpp/native-lib.cpp` shows how to use the provided libraries.

```
AHardwareBuffer* inAHB  = AHardwareBuffer_fromHardwareBuffer(env,inbuf);
AHardwareBuffer* outAHB = AHardwareBuffer_fromHardwareBuffer(env,opbuf);

JNI_GLOBAL::context->makeCurrent();

Simulation::HardwareBuffer inputBuffer(inAHB);
Simulation::HardwareBuffer outputBuffer(outAHB);
Simulation::EGLMap         inputImage(inputBuffer,   JNI_GLOBAL::context-
>getDisplay());
Simulation::EGLMap         outputImage(outputBuffer, JNI_GLOBAL::context-
>getDisplay());

outputImage.bindFBO();
```

The `HardwareBuffer` objects from the `ImageReader` and `ImageWriter` must first be converted to their native counterpart `AHardwareBuffer`. Android provides a JNI conversion tool that allows this to be done easily.

From there, API calls provided by the sample application are used.

The first step is to pass an `AHardwareBuffer` to a `HardwareBuffer` wrapper. Internally, this extracts a large amount of metadata from the buffer and makes that metadata available.

The next step is to pass the Android `HardwareBuffer` to an `EGLMap` of the sample application, which maps the buffer to an OpenGL texture. These steps are each done twice, for both input and output buffers.

The final step is to bind a frame buffer object for the output buffer so that the renderer can draw into the buffer with a fragment shader.

At this point, all of the necessary buffers are mapped, and the shader pipeline can be run to apply the edits to the frame. This edit application process has two steps. The first step runs the compute shader that applies the effects to the pixels and renders the output into a transfer texture. The compute shader cannot be used to render into the output buffer; therefore, a second step is required. This step involves a fragment shader that takes the output from the compute shader and renders it into the output `AHardwareBuffer`. The following code demonstrates how to do this.

```
JNI_GLOBAL::renderer->RunEditComputeShader(inputImage.getBufferTexture(),
                                           inputImage.getHardwareBufferWidth(),
                                           inputImage.getHardwareBufferHeight());

JNI_GLOBAL::renderer->render(JNI_GLOBAL::renderer->GetTransferTextureID(),
                             outputImage.getLutTexture(),
                             false,
                             false);
```

After this, the output data is in the target buffer. Calls for this procedure using the JNI are made in the `onFrameAvailable` method of the `FrameHandler` class. First, `onFrameAvailable` creates the two hardware buffers, as follows.

```
HardwareBuffer input  = inputImage.getHardwareBuffer();
Image outputImage     = writer.dequeueInputImage();
HardwareBuffer output = outputImage.getHardwareBuffer();
```

Next, `onFrameAvailable` calls the `processFrame` function. This function runs the C++ rendering code. After calling this function, the output image has data.

Finally, `onFrameAvailable` configures the dataspace and the time stamp. The parameter passed to `setDataspace` is the pipeline output dataspace, which varies for each piece of content.

For Dolby Vision profile 8.4 output, the specific settings to be applied are:

- `STANDARD_BT2020`
- `TRANSFER_HLG`
- `RANGE_LIMITED`

Setting up a decoder

To ensure an accurate preview experience for the user, the preview format needs to be adjusted depending on the target format. Dolby Vision is decoded differently if the target format for transcoding or editing is not Dolby Vision. Your platform will have its own mechanism for indicating which output format the user has requested. Dolby decoders must be set up differently depending on whether the output is in standard or high dynamic range. The steps to take depend on which platform the device is running. Refer to *Display management*.

For the DPU solution, the procedure is straightforward. The decoder outputs HDR by default, assuming that the output is Dolby Vision. For other cases like the GPU solution, the following code should be used. This code is an excerpt from the configure function in the file `app/src/main/java/com/dolby/capture/filtersimulation/CodecBuilderImpl.java`.

```
if (getSupportStatus() == PlatformSupport.DPU) {
    if
((decoderFormat.getString(MediaFormat.KEY_MIME).equals(MediaFormat.MIMETYPE_VIDEO_DO
LBY_VISION))
            && (decoderFormat.getInteger(MediaFormat.KEY_PROFILE) ==
MediaCodecInfo.CodecProfileLevel.DolbyVisionProfileDvheSt)
            && (outputFormat.equals(HEVC) || outputFormat.equals(AVC))) {
        Log.e("CodecSelection", "configure: Dolby Vision Decoder Filter Enabled");
        format.setInteger(MediaFormat.KEY_COLOR_TRANSFER_REQUEST,
MediaFormat.COLOR_TRANSFER_SDR_VIDEO);
    } else {
        Log.e("CodecSelection", "configure: Not enabling Dolby Vision Filter");
    }
} else {
    if (mime.equals(MediaFormat.MIMETYPE_VIDEO_DOLBY_VISION)) {
  c.setTransfer(Codec.TRANSFER_PARAMS[transfer]);
    }
}
```

This code first checks the target output format that the user selects.

Next, the requested output format is determined. If the request is for a SDR codec, then the decoder is instructed to downconvert to SDR before proceeding with editing. The first bold code makes this request to the decoder. If the request is for Dolby Vision output, this step is not necessary.

For the non-DPU solution, a transfer parameter must be passed, for both SDR and HDR output scenarios. The string value is ‘`TRANSFER_SDR`' for SDR output, and ‘`TRANSFER_DOLBY`' for Dolby Vision output. The second bold code applies this.

Testing

At this point, the HLG preview system should be working, and should be able to switch modes on the Dolby Vision decoders to get the desired output type. There is a simple test to verify that steps have been carried out correctly and that preview is running in HLG.

First, open the application and make sure a Dolby Vision video is playing. Also ensure that the output format is set to Dolby Vision. This is the only output format setting with which HLG output should be used. If the output is SDR, then do not label the frames as HLG.

To check the type of frame on the screen, follow these steps:

1. Ensure that the device is connected to a Windows computer by a USB cable.
2. Ensure that Android Debug Bridge (ADB) is installed on the machine.
3. Open PowerShell, and run the following command:

   ```
   adb shell dumpsys SurfaceFlinger | findstr 'layer:'
   ```

Running this command instructs `SurfaceFlinger` to report the format of all the layers currently on the screen, one of which will be the preview. The following output is produced by the sample application:

```
layer: 1575
name: MediaSync
z: 0
composition: Device/Device
alpha: 255
format: Y_CBCR_420_P010
dataspace:0x12060000
transform: 0/0/0
buffer_id: 0xb400007c27bb18d0
secure: 0

layer: 1572
name: VRI[MainActivity]#2(BLAST Consumer)2
z: 1
composition: Device/Device
alpha: 255
format:         RGBA_8888_UBWC
dataspace:0x00000000
transform: 0/0/0
buffer_id: 0xb400007c27bb5d70
secure: 0

layer: 1557
name: VRI[NavigationBar0]#3(BLAST Consumer)3
z: 2
composition: Device/Device
alpha: 255
format: RGBA_8888_UBWC
dataspace:0x088a0000
transform: 0/0/0
buffer_id: 0xb400007c27bb2a40
secure: 0

layer:  132
name: VRI[ScreenDecorOverlay]#0(BLAST Consumer)0
z: 3
composition: Device/Device
alpha: 255
format: RGBA_8888
dataspace:0x088a0000
transform: 0/0/0
buffer_id: 0xb400007c27bb5980
secure: 0

layer:  131
name: VRI[ScreenDecorOverlayBottom]#1(BLAST Consumer)1
z: 4
composition: Device/Device
alpha: 255
format: RGBA_8888
dataspace:0x088a0000
```

```
transform: 0/0/0
buffer_id: 0xb400007c27bb3970
secure: 0
```

In the sample application, the preview is the first layer displayed. It is critical to check that the dataspace value is `0x12060000`. This indicates that the frames are labeled as BT.2020 HLG limited range, which is needed for Dolby Vision 8.4 playback.

Using ST 2084 for Dolby Vision profile 8.4 HDR preview

HLG serves as one option for Dolby Vision profile 8.4 HDR preview, while ST 2084 (Perceptual Quantizer or PQ) offers an alternative HDR preview mode for the same Dolby Vision profile 8.4 content. Some implementations have selected ST 2084 due to its early adoption in Android HDR ecosystems.

HDR preview configuration with ST 2084

To implement ST 2084 as the HDR preview mode, modify the preview dataspace settings in the `/app/src/main/java/com/dolby/capture/filtersimulation/ImagePipeline.java` file as follows:

```
if (previewMode) {
    if (inputProfile == MediaCodecInfo.CodecProfileLevel.DolbyVisionProfileDvheSt
        && encoderFormat.equals(Constants.DV_ME)) {
        this.dataspace = DataSpace.pack(
            DataSpace.STANDARD_BT2020,
            DataSpace.TRANSFER_ST2084,
            DataSpace.RANGE_LIMITED
        );
    } else {
        // Alternative implementation
    }
}
```

When using ST 2084 for Dolby Vision HDR preview, a different look-up table (LUT) must be applied to properly map the content for display:

- The LUT resource is located at: `app/src/main/cpp/hlg_lut_1000_bt2020_pq_33.h`
- The LUT application is implemented in the shader: `/app/src/main/cpp/EditShaders.cpp`, which contains the implementation for loading and applying the ST 2084 LUT.

Trimming and remultiplexing

Trimming video directly on the encoded bitstream is a non-destructive process that preserves video quality. After trimming Dolby Vision content, remultiplexing is necessary to add the required Dolby Vision signaling to the MP4 container, ensuring proper identification and playback of the Dolby Vision format.

One of the most common editing operations is trimming or removing parts of a clip. The main objective of a trimming algorithm is to trim in a nondestructive manner. With few exceptions, the codecs used by mobile devices to capture content use a form of lossy compression. Consequently, every time a video is transcoded or edited at a pixel level, video quality is degraded.

Trimming differs from transcoding or editing in that it does not alter pixels. Therefore, an application can preserve video quality by trimming directly on the encoded video bitstream. This approach is demonstrated in the sample application. Instead of running baseband frames through an encoder, the sample application trims without loss by extracting encoded data from one file and copying it to another file just when the time stamp is within a certain range.

The Android `MediaMuxer` handles Dolby Vision content seamlessly across various operations. During these processes, the multiplexer automatically adds the necessary Dolby Vision signaling to the MP4 file. This signaling informs playback devices that the video is Dolby Vision content and should be interpreted accordingly.

6

Appendix

Quality control for Dolby Vision applications

It is important to confirm that all the video editing pipeline elements described in this documentation have been correctly implemented in your application. The following tools let you confirm that the output file is a valid Dolby Vision file.

For Dolby Vision bitstream objective quality control, please download one of the following verification toolkits:

- Dolby Vision Professional Verification Toolkit v3.8.0
- Dolby Vision Professional Verification Toolkit - Lite v4.3.0

For Dolby Vision bitstream in , HTTP Live Streaming (HLS) (HTTP Live Streaming), or MPEG Dynamic Adaptive Streaming over HTTP (MPEG-DASH) (MPEG Dynamic Adaptive Streaming over HTTP) quality control, please use the Dolby Stream Validator.

Dolby Vision technical information

- What are Dolby Vision profiles?
- How to signal Dolby Vision in ISOBMFF format (MP4)

Glossary

API
: Application programming interface. A set of functions that can be used to access the functions of an operating system or other type of software.

BT.2020
: ITU-R Recommendation BT.2020. A standard with a color space, for characterizing and encoding ultra-high-definition television.

BT.709
: ITU-R Recommendation BT.709. A standard with a color space, for characterizing and encoding high-definition television.

DPU
: Display processing unit. A hardware component of a Qualcomm system-on-chip (SoC) designed for accelerated processing of visual data represented as pixels.

GPU
: Graphics processing unit. A specialized electronic circuit designed for optimal use of memory and accelerated processing to create images in a frame buffer for output to a display device.

HDR
: High-dynamic-range imaging. A video imaging technology that captures, processes, and displays a significantly wider range of luminance levels compared to Standard Dynamic Range (SDR), resulting in enhanced picture quality with more realistic colors, deeper blacks, brighter highlights, and improved overall contrast.

HLG
: Hybrid log-gamma. High-dynamic range standard format developed jointly by the British Broadcasting Corporation (BBC) and Nippon Hoso Kyokai (Japan Broadcasting Corporation), and defined in ARIB STD-B67 and ETSI TS 101 154.

HLS
: HTTP Live Streaming. An adaptive streaming protocol developed by Apple for delivery of media content in various software environments.

IEEE
: Institute of Electrical and Electronics Engineers.

JNI
: Java Native Interface.

LUT
: Look-up table. In the context of Dolby Vision, a series of precalcuated display management data stored in a static text file that saves processing time and cycles for display management.

MPEG-DASH
: MPEG Dynamic Adaptive Streaming over HTTP. An adaptive bit-rate streaming protocol that enables high-quality streaming of media content over the Internet delivered from HTTP.

SDR
: Standard dynamic range. An ITU-R BT.709 signal with peak luminance of 100 cd/m^2.

SoC
: System-on-chip. An integrated circuit that integrates all components of an electronic system into a single chip.

YUV
: A file format that is encoded using the YCbCr color space.