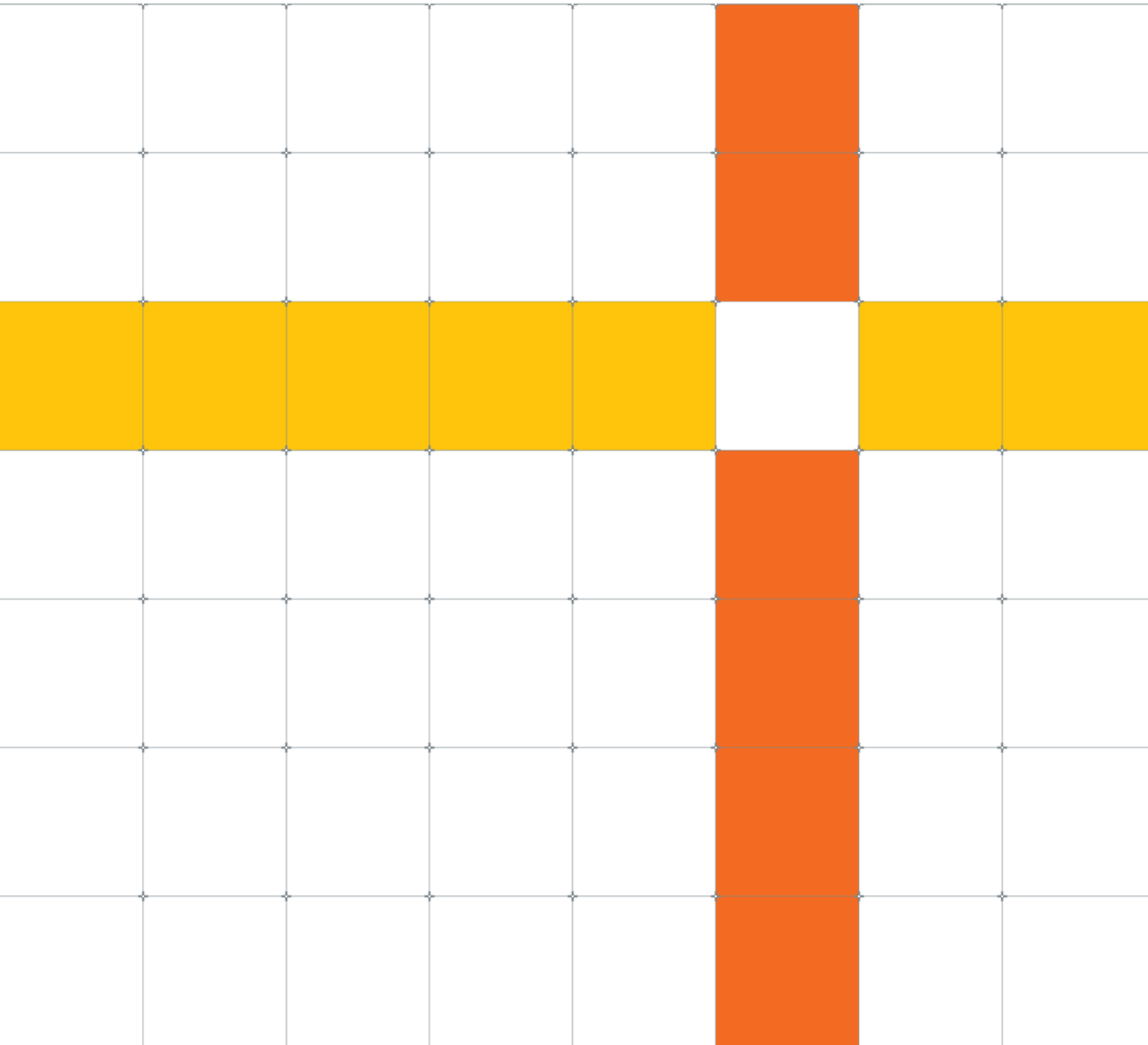

360-Degree Video Rendering
Using Arm Technology to Implement 360-Degree Video Efficiently

Panagiotis Christopoulos Charitos – Senior Engineer – Arm
Hans-Kristian Arntzen – Senior Engineer – Arm
Alberto Duenas – Senior Principal Video Architect – Arm
Daniele Di Donato – Staff Software Engineer – Arm

November 2017

Non-Confidential Proprietary Notice

LES-PRE-20349

Contents

1 Introduction

The introduction of virtual reality has brought new applications to the surface as well as derivatives of existing technologies. One improvement over existing technologies can be seen in the case of 360-degree video. The increased immersion of virtual reality can easily be applied to video - providing superior user experience over the traditional video that is projected into flat surfaces.

For any emerging technology, the first step of evolution is to adapt existing techniques and knowledge to the new paradigm. For 360-degree video, the evolution begins from adapting existing planar video technology and projecting it to virtual three-dimensional environments. Video is an integral part of everyday life and countless man hours were spent to optimize it. Basing 360-degree video on existing technology is the first step of evolution.

Improving the experience and efficiency of 360-degree video requires solutions specifically tailored to this use case. Organizations like MPEG are working on creating new standards specifically designed for 360-degree video. This includes; view dependent video encoding and delivery, improvements on compression and encapsulation, new extensions, optimal projection methods and other technologies that are needed for the efficient processing and delivery of 360-degree video.

This whitepaper describes how existing Arm technology can be used to implement 360-degree video efficiently, and how the different hardware and software components interact with each other in order to produce high quality final results.

2 360-Degree Video Decoding Data Flow

Rendering 360-degree video efficiently touches many parts of a mobile platform, and needs cooperation from three important IP blocks to enable efficient high-resolution video at high framerates, they are:

1. Video Processing Unit (VPU). The VPU is responsible for decompressing the incoming video stream to YUV frames, a format which the texturing unit of the GPU can read.
2. Graphics Processing Unit (GPU). The GPU is responsible for reading the 360 degree video and projecting it down to a 2D image which can be displayed on a display. The GPU will take into consideration where the user is currently viewing (viewport). For VR, there are additional transformations that need to happen before the final projection into the screen, those include lens distortion and chromatic aberration. In addition to this, the GPU may be producing additional graphics and UI elements that will be merged with the video before it is displayed. Over time these additional transformations are expected to move into the Display Processing Unit and that will translate into higher performance and lower energy consumption.
3. Display Processing Unit (DPU). The DPU is responsible for scanning out the final image from memory and sending it to the display. The DPU may also perform framebuffer decompression and YUV-to-RGB conversion while scanning out.

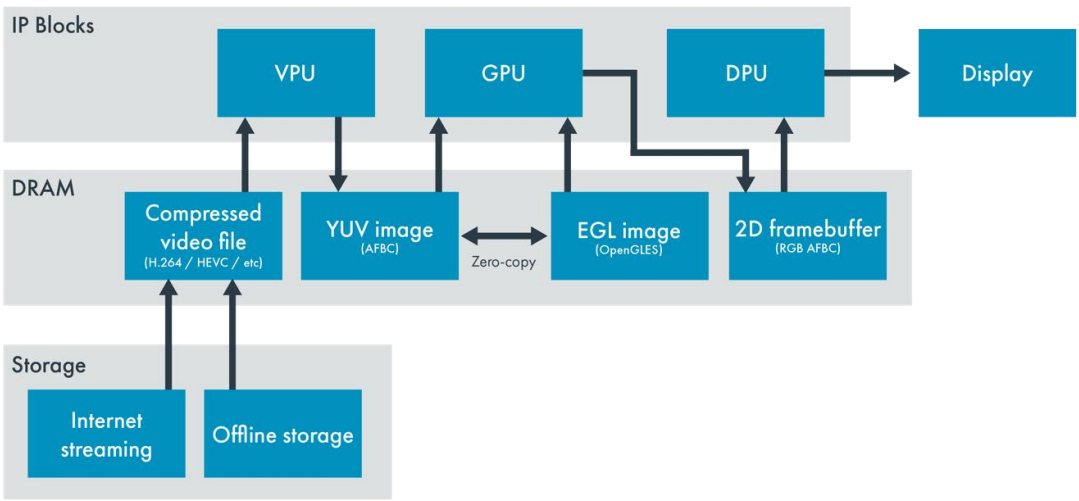

Figure 1: 360-degree video decoding flow

In traditional (2D) video the GPU is not required and the video frames may be converted from 4:2:0 YUV to RGB by the DPU directly. For 360-degree video however, the GPU will have to perform this conversion since the texels of the video frame will have to undergo a projective transformation to be correctly mapped to a geometry that can later be visualized on the screen. In all Mali GPUs, converting YUV to RGB is an implicit conversion that happens in two steps. When a shader is sampling the video frame, the texture unit reads and returns a color in YUV space, then some arithmetic instructions inside the shader convert it to RGB implicitly (it is possible to get YUV without any conversion as is explained later in this whitepaper). Sampling YUV 4:2:0 textures requires

sampling data from multiple color planes, which before Mali-G51 would consume multiple cycles in the texture pipe. From Mali-G51 onwards, planar YUV can be sampled at same rate as RGB textures.

3 Android Media Subsystem

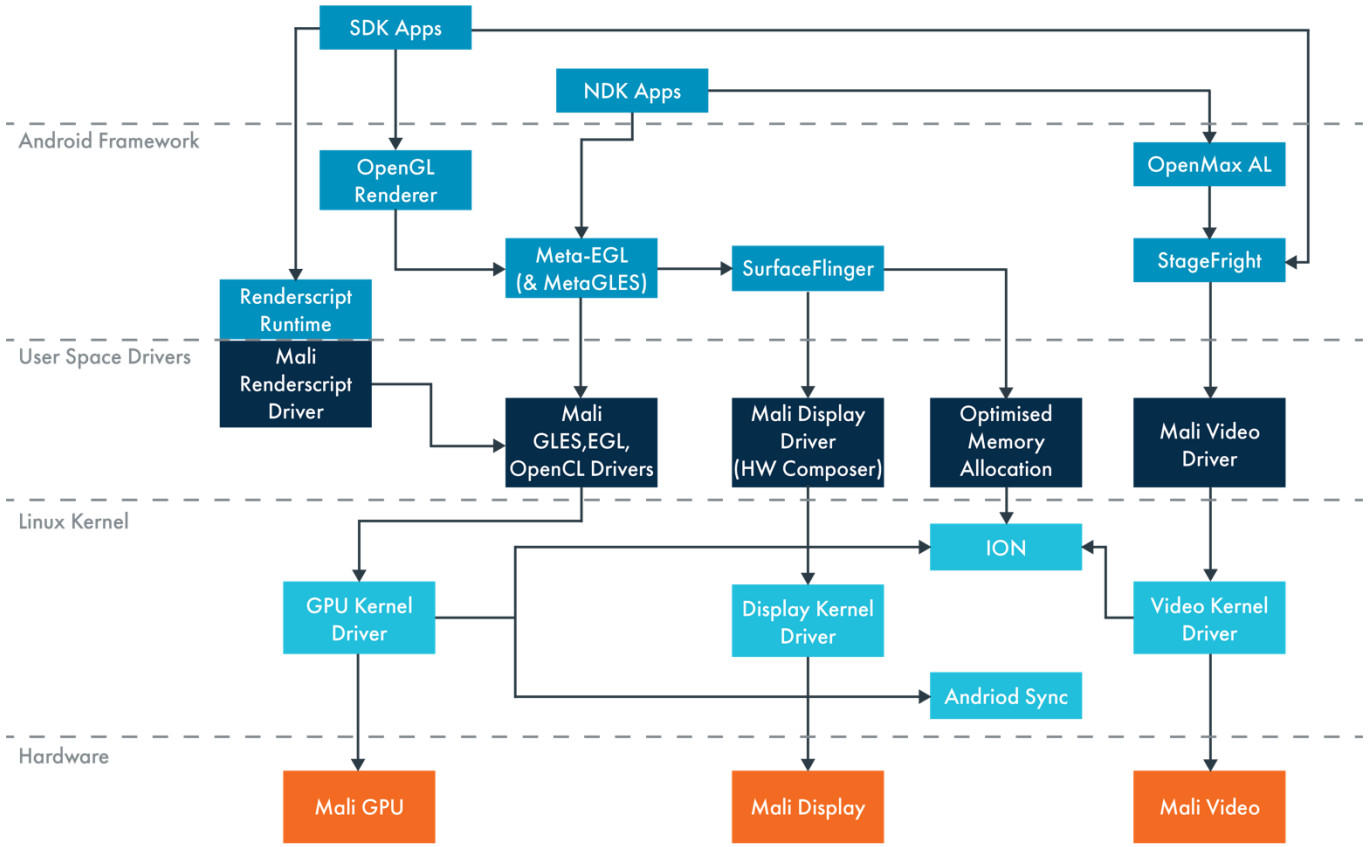

Figure 2: Android Multimedia Software Stack

The Android media subsystem abstracts away multiple video codecs behind multiple layers of abstractions. From the application point of view, the media framework can decode videos, and provide you an EGL/OpenGL ES external surface which applications can use to render to screen as they please.

The media framework on Android calls down to Stagefright, which is hidden from applications. Stagefright framework then interacts with multiple codecs. These codecs are implemented using the OpenMAX integration layer. OEMs can integrate hardware codecs this way, and declare that certain codecs and formats are supported through hardware. Google also provides software fall-backs when hardware codecs do not support certain formats.

To facilitate zero-copy buffer sharing, OpenMAX implementations can take an EGL image handle which a video codec can use as the output buffer for video. Therefore, the VPU and GPU can work together with no extra copying step.

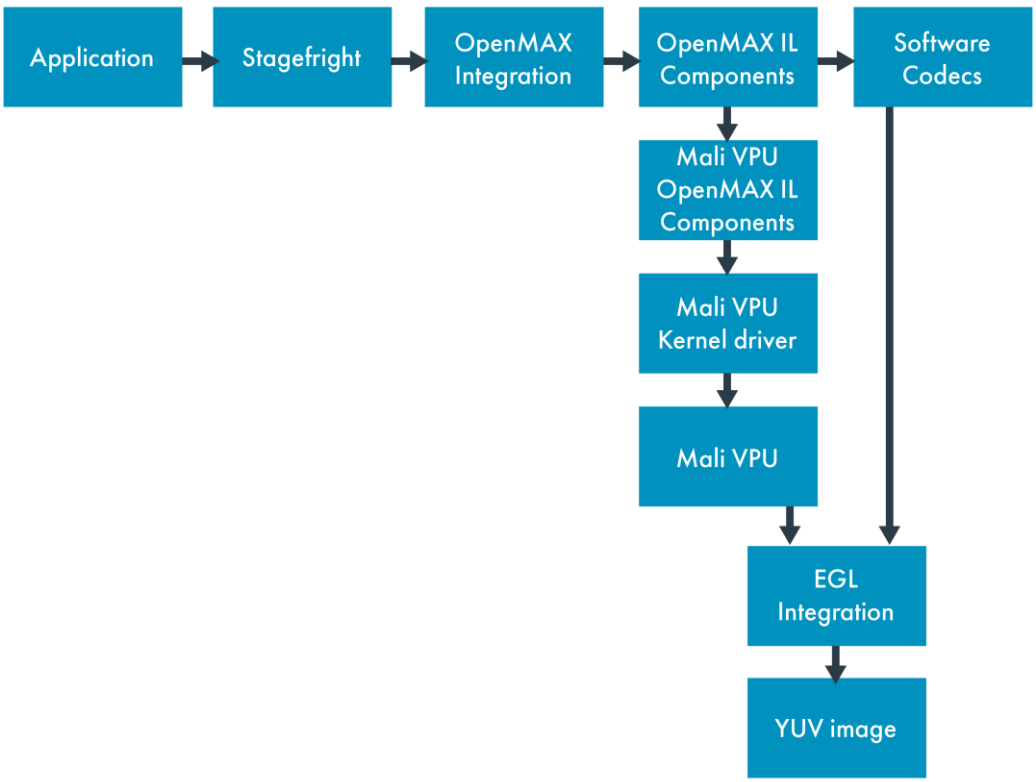

Figure 3: Android media subsystem

4 Rendering 360-degree video

Android media subsystem abstracts the video decoding process. The application can only control how to display the decoded video frame into the screen. In 360-degree video's case the GPU will have to project the video frames into 3D space before handing them to the display controller. Interfacing with Android is done using EGL external images which wraps the native memory buffers of the decoded frame and can be accessed by OpenGL ES. Since the most common color space for video is some form of YUV the OpenGL ES implementation supports reading from YUV images as well. The required OpenGL ES extension that adds support for external EGL images can be YUV as well as GL_OES_EGL_image_external, which are already supported in all Mali GPUs. GL_OES_EGL_image_external is an extension that works with OpenGL ES 2.0 onwards. For OpenGL ES 3.0 a similar extension GL_OES_EGL_image_external_essl3 is supported on Mali GPUs which support OpenGL ES 3.0.

The video frame is a two-dimensional surface that needs to be "wrapped" to 3D geometry. Whilst it can be captured using 3D capable equipment, it still needs to be projected to a series of 2D video frames. The GPU program will have to apply the inverse projection to move from 2D back to 3D space. There are a number of projections with the most popular ones being equirectangular projection (ERP) and cubemap projection and more complex examples like the icosahedron projection.

4.1. Equirectangular projection (ERP)

Equirectangular projection is a concept used in cartography that maps the earth and a sphere into a two-dimensional map. The horizontal coordinate in the map is the longitude and the vertical coordinate is the latitude. One obvious disadvantage with the equirectangular projection is that the projection area is not uniform over the image. In 360-degree video this can be observed as higher texel density on the poles and lower in the rest of the sphere. An additional disadvantage is that the more we move to the poles the more distorted the projected output becomes. An advantage of equirectangular projection is that it is quite simple to encode and transmit since the locality of data works in favour of bit rate.

Figure 4: Video frame in equirectangular projection

Figure 5: Equirectangular video mapped to a sphere as seen from outside the sphere

For the equirectangular projection to work the video needs to be encoded or captured accordingly. When visualizing the video on a device, the fragment shader converts the spherical coordinates to two dimensional planar coordinates that will be used to sample from the external EGL image. The GPU will convert from YUV color space to RGB implicitly and the shader will output that RGB result to a framebuffer.

The conversion of spherical coordinates to planar texture coordinates is a relatively expensive computation but can be avoided altogether in run-time.

The computation needed to convert a normalized world space direction (3D, x/y/z) into a texture coordinate (2D, u/v) is roughly:

```
u = 0.5 + 0.5 * (atan2(x, z) if |z| > epsilon else sign(x) * PI * 0.5) / PI
v = acos(y) / PI
```

In a naïve implementation, u and v are computed per pixel, which are then used to sample the video texture. A more optimal approach is to project the video to a pre-generated sphere where the planar texture coordinates are already calculated and baked into the vertices of that sphere. In our experiments, we used a sphere that consists of multiple patches and every patch is a square grid of 32x32 quads which are drawn as a triangle strip. The mesh is modelled as a cube, which is then compacted into a sphere by normalizing all positions. For each

cube face (6) a grid of 8x8 patches are laid out. The reason to split the sphere up in multiple patches is that we only need to perform vertex shading on the patches which are visible from the camera. For each vertex, we compute the normalized position x/y/z and compute u and v as mentioned above. When using this method however, we must be aware that u can wrap around. atan2() has multiple valid solutions, and when interpolating over a primitive we must make sure that all atan2() solutions for each vertex are in the same quadrant.

The fragment shader will just interpolate a varying which contains the planar texture coordinates and use those coordinates without the need for any kind of expensive coordinate translation. Effectively, this implements a piecewise approximation of the u/v transform and the accuracy of the interpolation depends on how dense the vertex grid is. The proposed numbers have been shown to work well in practice.

It is important to note that the exact scheme for how to create the sphere mesh is not the most vital part. The important part is that each vertex pre-computes u and v and passes those coordinates down to the fragment stage.

The application only needs to generate the sphere and bake texture coordinates upfront. The next step is to position the camera at the centre of the sphere and render it using appropriate view and projection matrices.

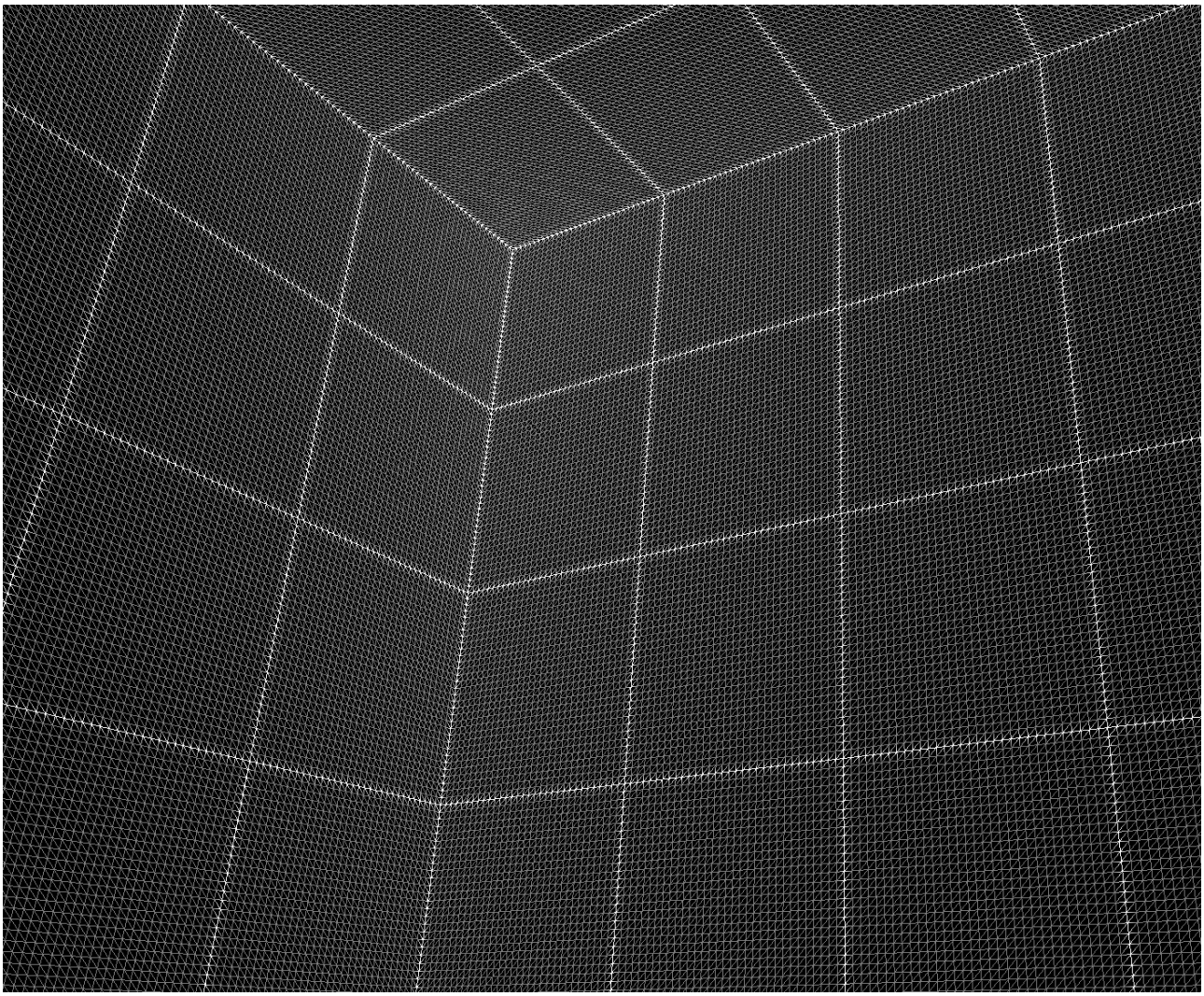

Figure 6: Visualizing the sphere's patches

Note that even if the mesh itself is a sphere in Figure 6, it was created as a cube, so in this wireframe representation it looks like one, because from the inside, normalizing the cube coordinates does not change the direction to the vertex in question, but from a perspective correct interpolation point of view, it is significant to normalize. In Figure 7, we see how the mesh looks from the outside. The 32x32 subdivision was reduced to 8x8 in this image to make the wireframe more visible.

It is important to note that creating this sphere as a cube is not necessary. Any well-subdivided sphere mesh will work. A cube was used here due to its simplicity.

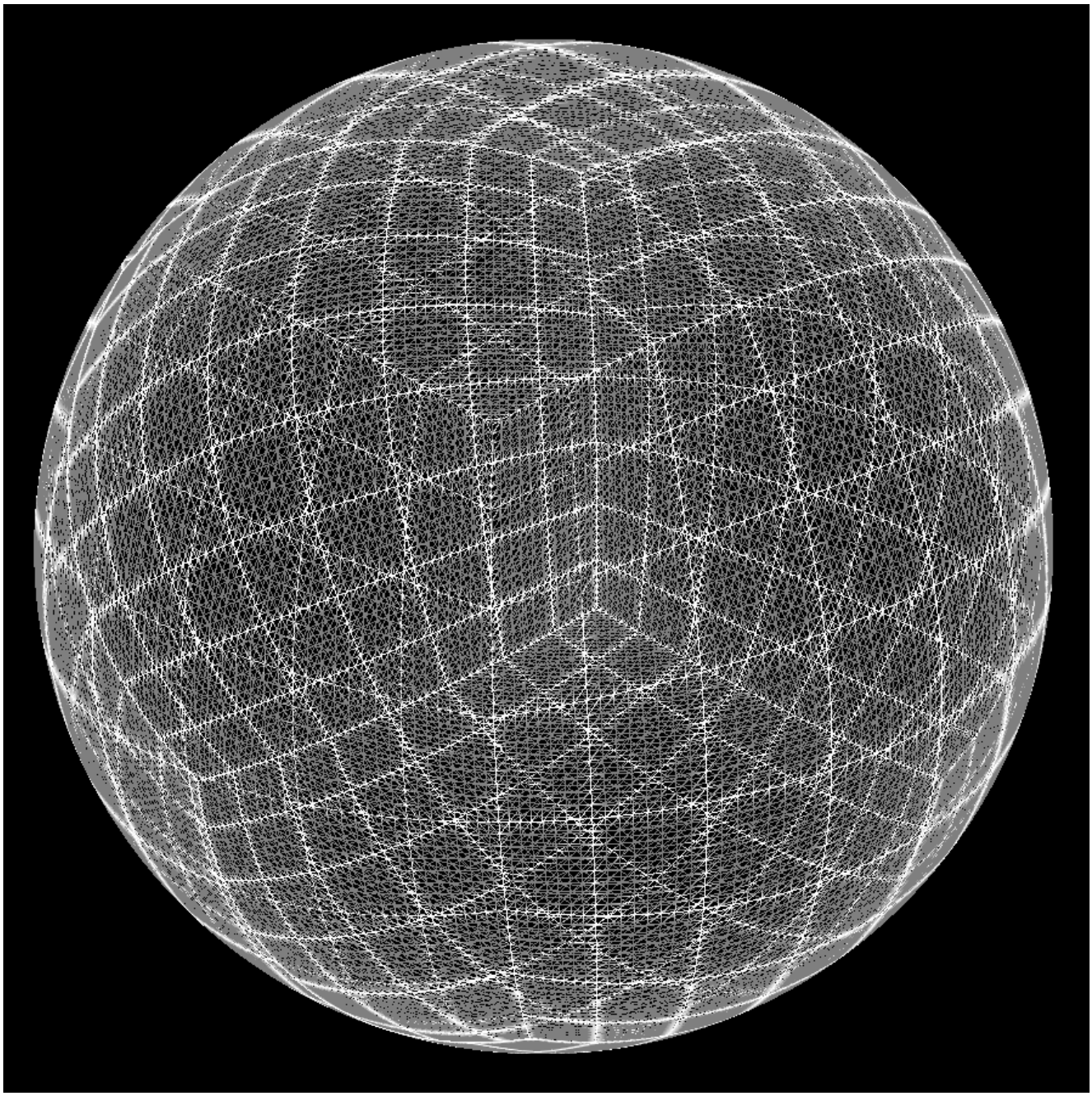

Figure 7: Sphere mesh seen from outside

4.2. Cubemap projection (CMP)

In cubemap projection, the environment is mapped into the six sides (also known as faces) of a cube. Cubemap projection may be preferred over simpler methods, like equirectangular projection because it eliminates the distortion in the poles and it provides a more linear and view independent result. Current state of the art video codec are all block-based video codecs and rely on motion estimation on a per-block basis to greatly reduce bandwidth in videos. Cubemaps are therefore quite favourable due to their tendency to not warp the projected video unnecessarily.

Figure 8: A computer generated video frame with 6 faces (layout: +X, -X, +Y, -Y, +Z, -Z)

First, we need to provide some details around cubemap support on Mali hardware and software. Normally cubemap rendering with OpenGL ES is pretty straight forward. Three dimensional coordinates are translated into two dimensional coordinates plus a face index. This conversion is abstracted by the GLSL or SPIR-V shading languages. One problem with this translation is when sampling close to the cubemap edges or corners using bilinear filtering. The pictures below describe this problem. When rendering one face of the cubemap as a plane, its texture coordinates will cover one of the faces in the video frame (for example +X).

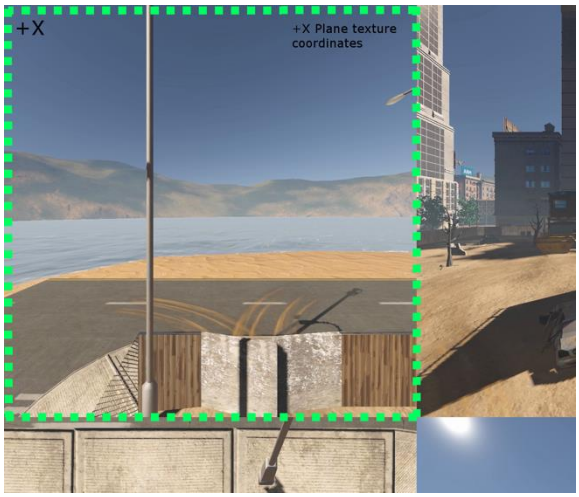

Figure 9: Texture coordinates for +X plane

When approaching the edge of the plane we are reaching the edge of a face inside a single video frame and that means the neighbouring pixels are the ones from another face (-X in this case).

Figure 10: Edge between face +X and –X

Due to bi-linear filtering, when the texture coordinates are in-between pixels (the red dot in the picture below), the color sampled on the edges will be a blend of the 2 edge faces that represent different views of the video with non-coherent information.

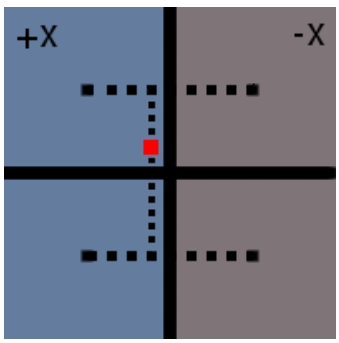

Figure 11: Bi-linear interpolation of neighbouring pixels belonging to different faces

To have the optimal visual result we need a way to filter smoothly between faces, either through hardware support or software techniques.

Seamless cubemap filtering is a GLES and Vulkan feature which allows the hardware to use texel data from two or three cube faces when sampling at a cube edge or corner respectively. Mali hardware supports seamless cubemap filtering for regular RGB(A) textures in hardware, but not for YUV ones.

In addition to the lack of seamless YUV cubemaps, the software stack (through GL_OES_EGL_image_external extension) doesn't support textures in a cubemap arrangement, only 2D. Seamless YUV cubemap filtering is not possible in hardware at the moment but we show that a software solution is just as good further on.

The first approach for seamless cubemap filtering is manual edge and corner filtering in YUV space. The video is encoded that way so that each frame is divided into six sub-regions, each one of them containing one face. The exact way this video is laid out is not vital as long as the 360-degree video system knows about the layout. Here we have chosen the layout (+X, -X, +Y, -Y, +Z, -Z) packed in a 3-by-2 grid.

Since the output from the video decoder is a big 2D image, the shaders which run on the GPU need to take this layout into account without extra copying or conversion stages. In Android for example, the extra copy is avoided by making use of Gralloc for allocating the memory which will be used by the VPU to decode video frames into, and the GPU to read the frames from. The conversion to YUV to RGB is done automatically by the shader compiler in the Mali-TXXX, G5x and G7x series. In the Mali-Gxx series, this conversion is further optimized to give the best execution time compared to custom conversion code.

4.3. Manual seamless cubemap filtering

In the first method we present, we take the YUV image as input from the video decoder and render an RGB image which can be displayed on-screen. In Figure 12, we see green pixels which need special edge filtering. In the corner, we see a few red pixels which need to perform cube corner filtering. As a result the visual output has no seams, just as if being rendered with seamless filtering in hardware.

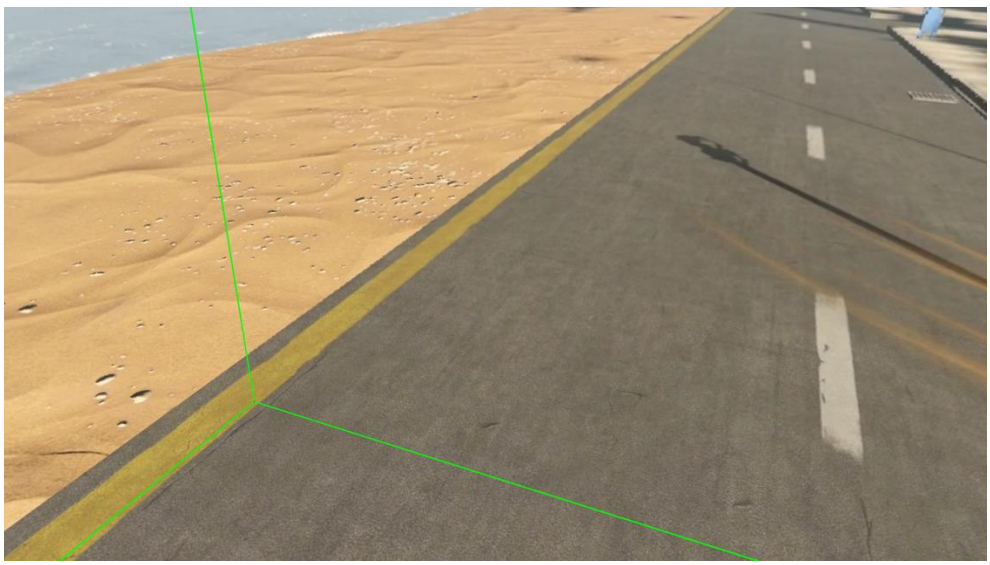

Figure 12: Pixels which need special shading

In order to do this efficiently we make use of the GPUs abilities to:

- Rasterize primitives with perspective correction.
- Interpolate YUV textures with bilinear filtering.
- Convert YUV to RGB.

We can render the output frame by rendering the 6 cube faces as planes in 3D space using OpenGL ES. We can then map each cube face to their respective plane. As we already mentioned, just rendering 6 planes as-is will give artefacts at the seams because near the edges, the bilinear interpolator will attempt to sample unrelated image data. The way we solve this problem is to run a special shader for the few pixels that are close to an edge and perform manual interpolation between two different faces. This allows us to hit the fast path when we are inside the cube face (99.9% of the pixels), and have a slightly more expensive shader when we need to. The special shader needs to know about the adjacent faces, and the shader will sample the faces individually and blend the result together in a similar fashion to how the hardware would do it.

Manually filtering YUV samples requires a separate OpenGL ES extension, GL_EXT_yuv_target, which depends on OpenGLES 3.0, which limits the support to Mali-T600 series and later GPUs. The purpose of this extension is to avoid the implicit YUV to RGB conversion which GL_OES_EGL_external_image does. We will convert YUV to RGB manually after completing the filtering. Note that manually converting from YUV to RGB does not lose performance as the shader compiler needs to perform this conversion.

4.4. Video encoding with borders

In a second approach, the problem of edge filtering manually between cube faces is moved from runtime to the video encoding phase. The video frames will still contain the 6 faces but with a border around them. The borders around each face contain texel data from the neighbouring faces. Rendering is similar to the previous approach with the main difference being that there is no manual edge and corner filtering. The cube is rendered as six quads, and each quad will have texture coordinates that don't touch the borders. That way the hardware texture unit will sample neighbour texel data from the artificial borders in order to compute the bi-linear filtered texels.

One interesting observation out of this approach is that texel data is duplicated inside the video frame. Since the video compression is lossy nobody can guarantee us that the duplicated texels will have the same color when decoded. This might create undesired results around some edges and for some video frames. This effect is amplified when the video quality decreases and can be mitigated if we increase the size of the border. Technically, a single pixel border is enough for high quality video but the border can easily be increased to eliminate such artefacts.

Replicating the edges of the borders is a straightforward blit operation however, the corners require a different approach. Each face contains four corners and each corner's color can be the average of the three neighbour texels. Experimenting with two pixel wide borders showed that the average of the three will look good enough as well. One interesting enhancement with this approach is to use bi-cubic filtering instead of just bi-linear for higher quality video. The increased size of the border can accommodate such an enhancement.

4.5. Conclusion on 360-degree video projections

Equirectangular projection is the most common way to represent 360-degree video and is well supported by social media and streaming services. Its use is widespread and a lot of use cases have been made thanks to the ability to run this type of projection on current devices. Unfortunately, this method doesn't give the best quality for the resolution and bandwidth used, yet the video and standardization bodies are working on more efficient and effective projections methods. Cubemap projection is one projection that improves the delivery of the 360-degree videos. Our techniques prove that is viable to use this type of projection with current hardware and small implementation changes inside the video creation. In the future these modifications may not be required as new extensions could allow this work to be done automatically without developer intervention.

5 Choosing the right IP

HEVC is currently the most efficient video codec as it offers higher quality per power consumption ratio. The industry and especially MPEG are working on specialized formats for 360-degree video thus; new codecs are expected to change the landscape in the near future, but until those materialize, HEVC would be the preferred compression format.

Decoding high pixel count video is a quite demanding task thus we recommend a Mali-V61 codec in MP8 configuration to be able to decode 8K and combined this with Arm lossless compression scheme (AFBC) as this needs high memory bandwidth from the from external memory. To put that into perspective, decoding a YUV 8K 10 bits video translates to 7680×4320 x 60 x 10 x 1.5 = **3.73248 GB/s** to write and read at 60 fps. By using AFBC we can reduce that by up to 50%. For RGB 8K 10 bits the bandwidth is 7680×4320 x 60 x 10 x 3 = **7.46496 GB/s** to write/read at 60 fps. That number doesn't include the cost of reading and writing by the GPU (that depends on the resolution and lens distortion & chromatic aberration methods).

There is very little computation power needed except for performing YUV to RGB conversion in the GPU. Mali-G5x greatly improved YUV texturing performance over previous generations, and is able to sample YUV at the same rate as RGB. From a performance per area consideration in 360-degree video, Mali-G5x stands out when targeting high resolutions.

6 A glance to the future of 360-degree video

The more technology integrates seamlessly with human interaction the more is possible to exploit some particularities of human physiology to improve the technology itself and push it forward. For 360-degree video we can exploit the human field of view (viewport) to reduce the amount of computation executed to decode and play each frame. In fact, when playing a 360-degree video on the headset only a small portion of it is visible to the user at a given time. The rest of it is not visible but the streaming service had to push the whole frame over the network and the decoder had to decompress the entire frame to actually show just one small portion of the whole frame. View dependent streaming and decoding is an area of intense development where companies and standardization bodies are looking to reduce the large bandwidth that will be needed for the delivery of future 360-degree videos. The overall concept behind it is really simple and was hinted in the previous sections as well. If a frame can be subdivided into tiles, only the necessary tiles are decoded by the video decoder. The VR headset orientation will act as a sliding window within the frame and that pose can be used to send data to the VPU as a mask and only the tiles that are needed to be decoded.

The concept can be pushed even further up the chain to the video streaming provider. In network with non-sufficient throughput, this method can be used to speed up playback time. Instead of buffering a lot of 360-degree frames that are then not displayed entirely, buffering can be limited to a smaller number of tiles so that

the pose information will still be valuable (head rotation speed is limited) when the frame is going to play back and it can be used to request only a subset of the tiles. This will save both bandwidth when transmitting the video as well as allowing lower end systems to provide 360-degree video.