从手游到VR

《Pico视频》渲染管线优化经验分享

邹春毅
《Pico视频》引擎团队负责人

ByteDance字节跳动

Pico视频渲染技术栈

PICO 视频

开播端
- ODS (实时全向3D渲染)
- 6Dof动作捕捉
- 多端开播工具
- ......

Web

Cloud
- 云渲染
- 云端超分
- 8k+VideoEncode
- Depth/Matting from Image

Web

OS
- CVController
- TrackingService
- StationService
- GetHeadSensor
- PvrDisplayInter
- ......

SDK

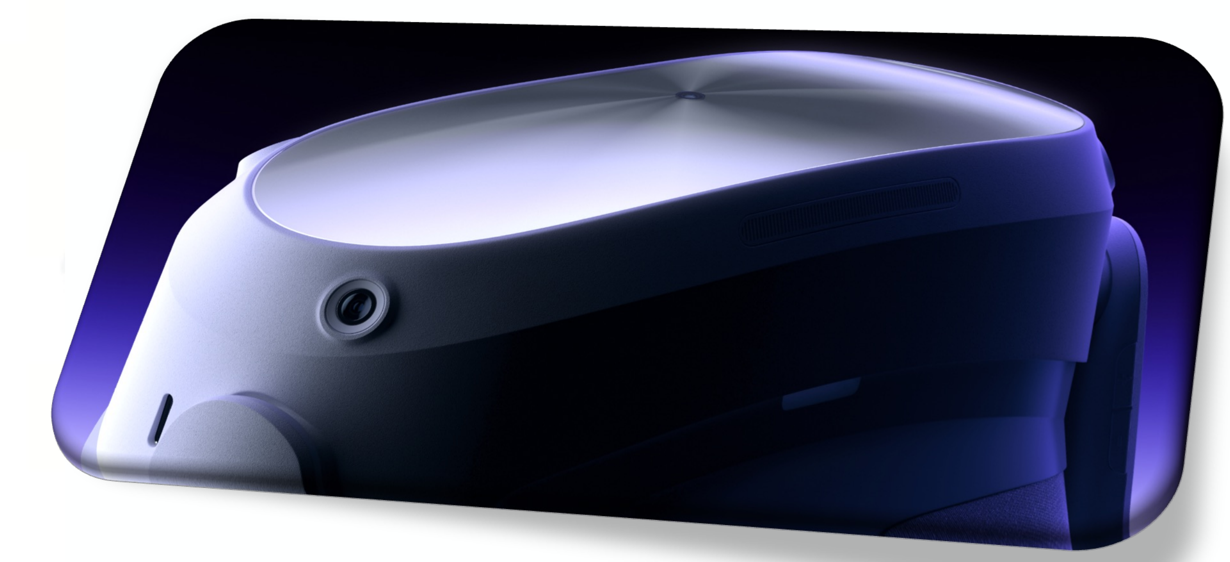
Unity
- VRS (可变速率着色)
- DirectOSE/YUV
- 颜色空间转换
- External RenderTexture
- SubPass
- WebXR
- GPUDriven
- ERP/CMP/EAR投影
- PBR/HDR/场景效果/卡渲.....
- ......

Context

Android
- Native渲染
- 本地超分
- External透传
- TileBasd 8k+VideoDecode
- DepthDecode
- ......

Eyebuffer

Runtime
- Compositor
- ATW/AppSW
- 锐化/超分
- ......



ByteDance字节跳动

郜春毅 3612

主流的硬件基数

续航保持在2.5小时，意味着每小时的平均功耗控制在2100mAh左右。

目前主流的VR设备采用骁龙XR2Gen2，发布于2019年，基于骁龙865改版。

5300 mAh
High capacity battery

Color See-through

Snapdragon 骁龙 XR2 第一代
Qualcomm® Snapdragon™ XR2

Ultrawide Pancake Optics
105° ultra-wide field of view

Motorized Interpupillary Distance Adjustment
62-72mm inter-pupillary distance adjustment

PICO 4

Integrated High Fidelity Speakers
Spatial audio

6DoF
Space Positioning
Sub-millimetre positioning accuracy
Millisecond tracking speed

4K+
Super-vision Display
4320 × 2160

Face Cushion

HyperSense
Vibration Controllers
Vibration frequency ranges from 50Hz to 500Hz

All-in-One Lightweight Design
Balanced design, easy to wear

物理分辨率在(单眼2160x2160，双眼4320x2160)，渲染分辨率最低通常在1504x1504，Pico视频通常在1804*1804，渲染像素是1080p的1.57倍

PICO视频
ByteDance字节跳动

PICO视频

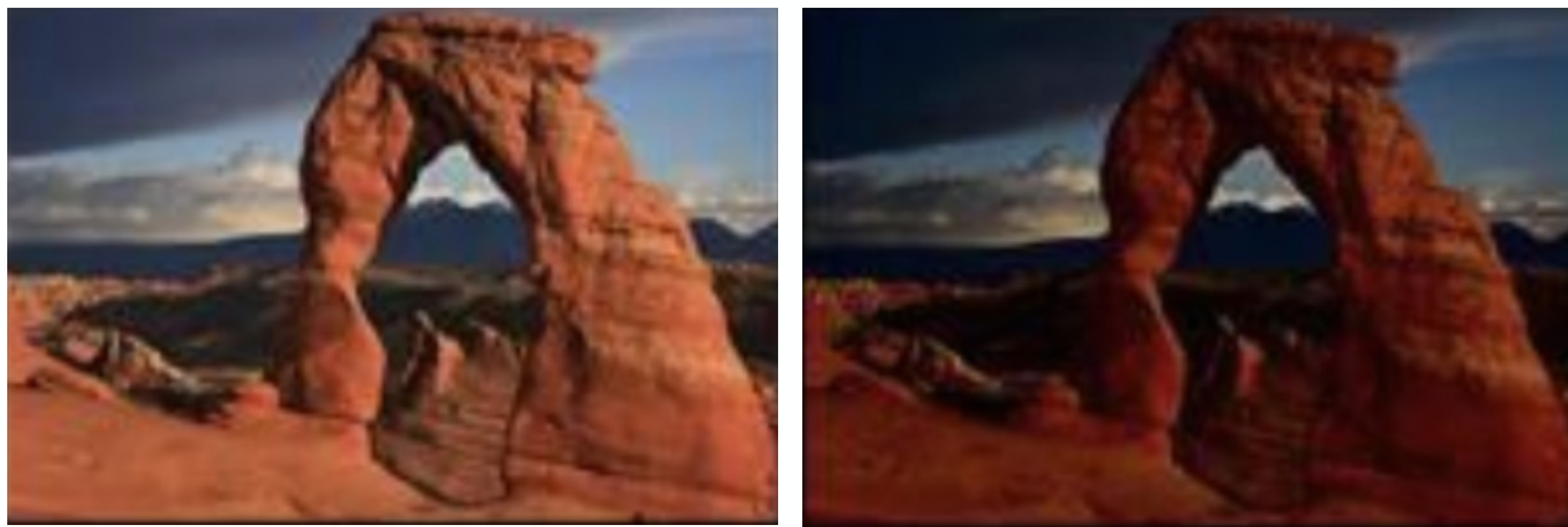
颜色空间转换

ByteDance字节跳动

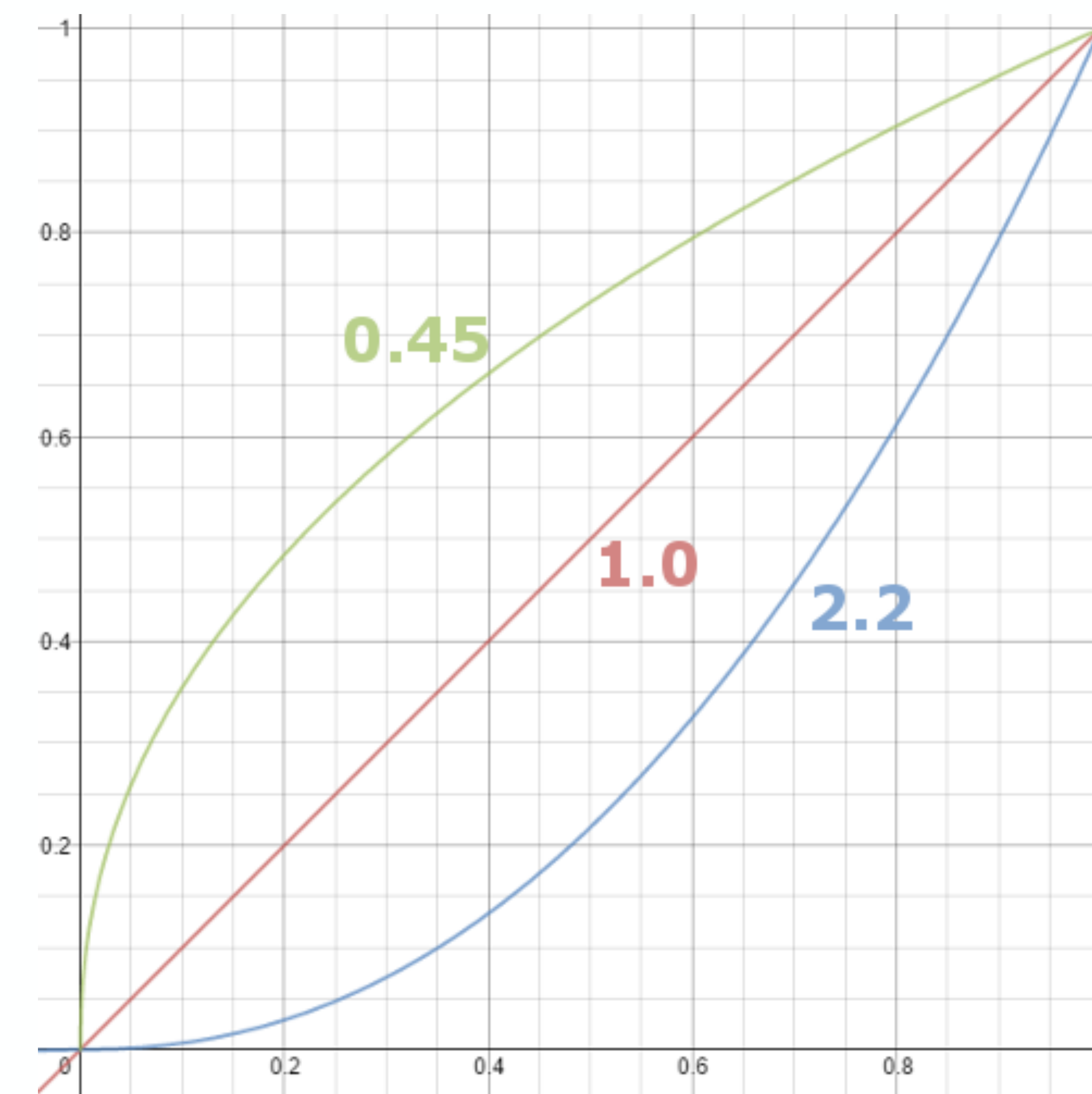
颜色空间转换

为什么要使用Linear空间

在物理世界中，如果光的强度增加一倍，那么亮度也会增加一倍，是一种线性的关系。而历史上最早的显示器显示图像的时候，电压增加一倍，亮度并不是增加一倍，是一种非线性关系。亮度增加等于电压增加的2.2次幂的非线性关系。这个2.2就叫做显示器的Gamma值。



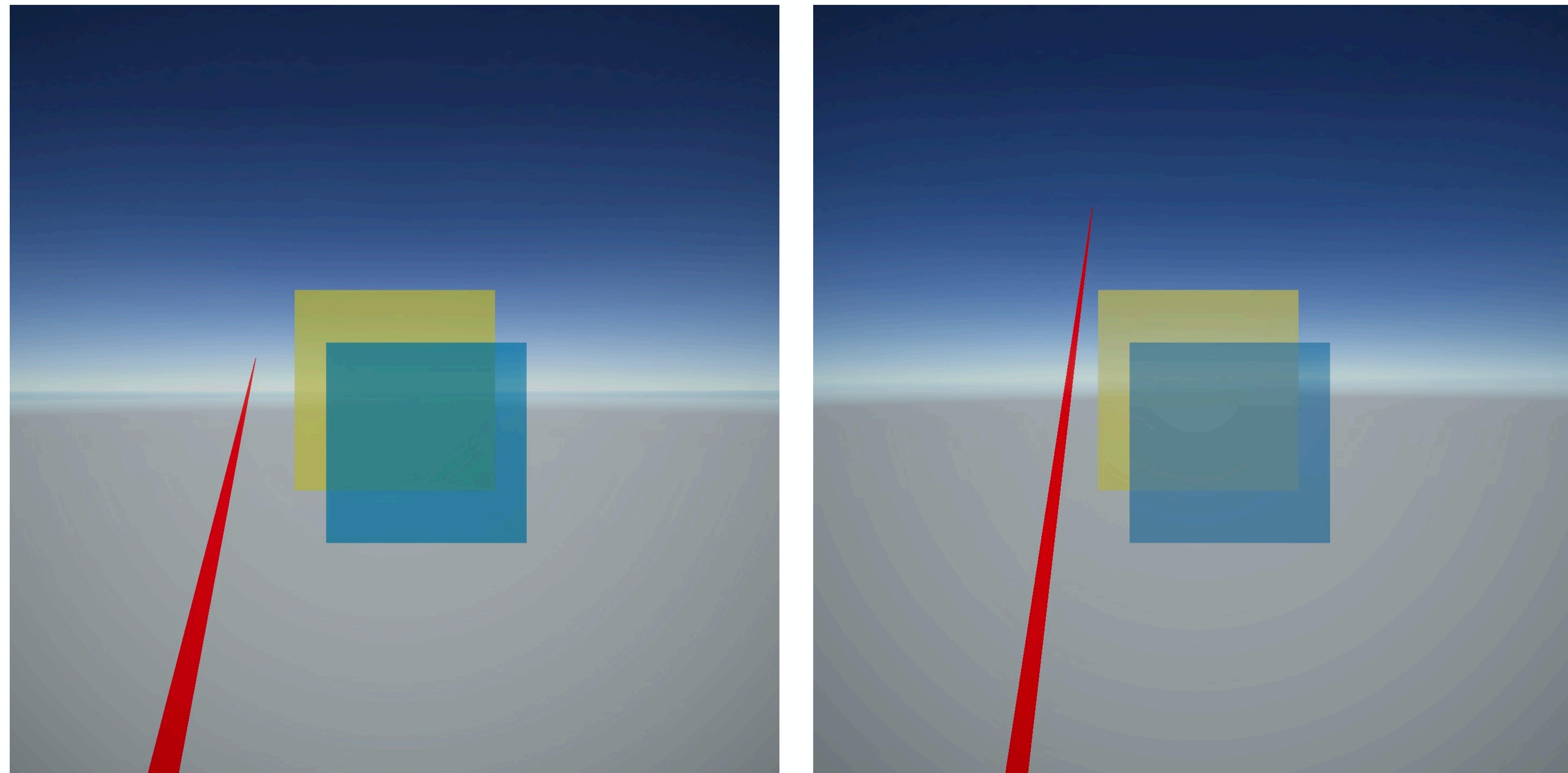
（左边为现实世界基于线性空间的图像，右边为显示器输出的图像）



PICO视频

颜色空间转换

UI半透明混合面临的问题



(左侧为Gamma空间下的混合结果，右侧为Linear空间下的混合结果)

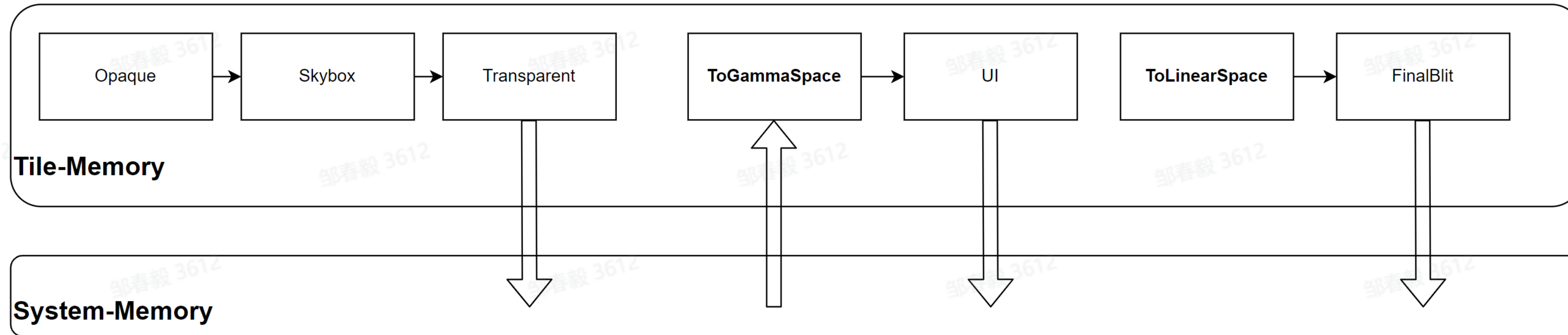
Gamma空间下的混合公式：
color = (A.rgb * A.a) + (B.rgb * (1 - A.a))

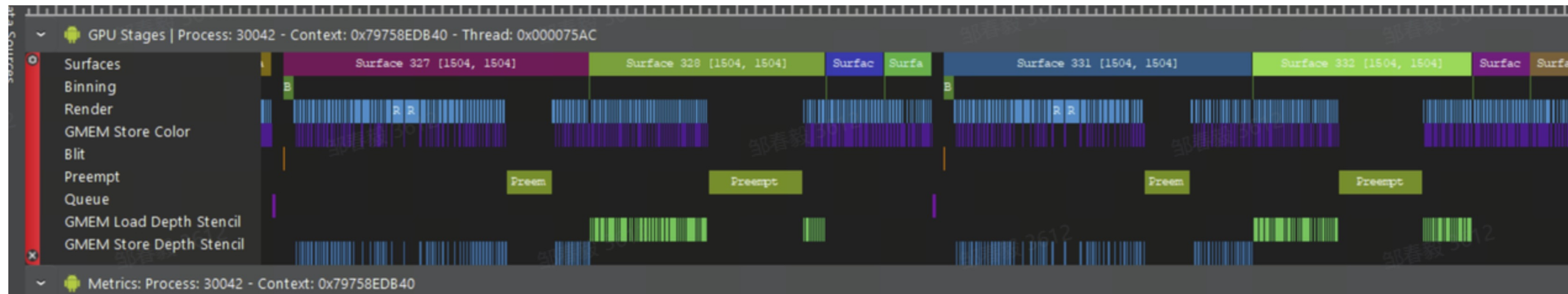
Linear空间下的混合公式：
color = ((A.rgb ^ 2.2 * A.a) + (B.rgb ^ 2.2 * (1 - A.a))) ^（1 / 2.2)

ByteDance字节跳动

颜色空间转换

手游常规的解决办法



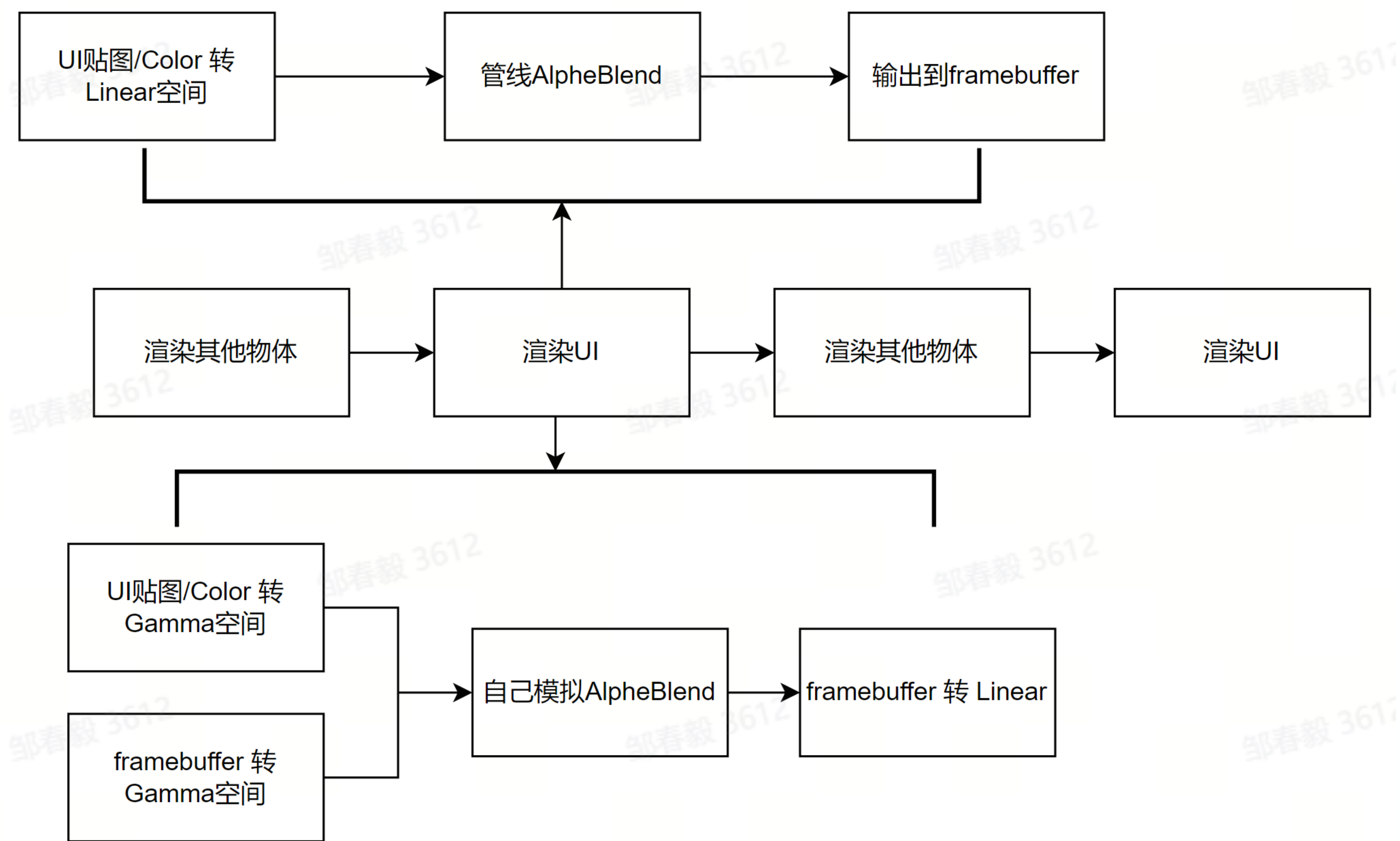


颜色空间转换

基于GL_EXT_shader_framebuffer_fetch扩展的解决方案

https://registry.khronos.org/OpenGL/extensions/EXT/EXT_shader_framebuffer_fetch.txt

开启GL_EXT_shader_framebuffer_fetch后，当前像素值将变为可读写（基于Tile-Memory），自己在pixel中模拟混合公式，转换到Gamma后混合，再转换到Linear输出。



(上半部分为默认的渲染流程，下半部为基于GL_EXT_shader_framebuffer_fetch的渲染流程)

与MASS的兼容：
对于开启MSAA的情况，FrameBufferFetch获取的Buffer数据为自动resolve之后的结果，无需手动处理。

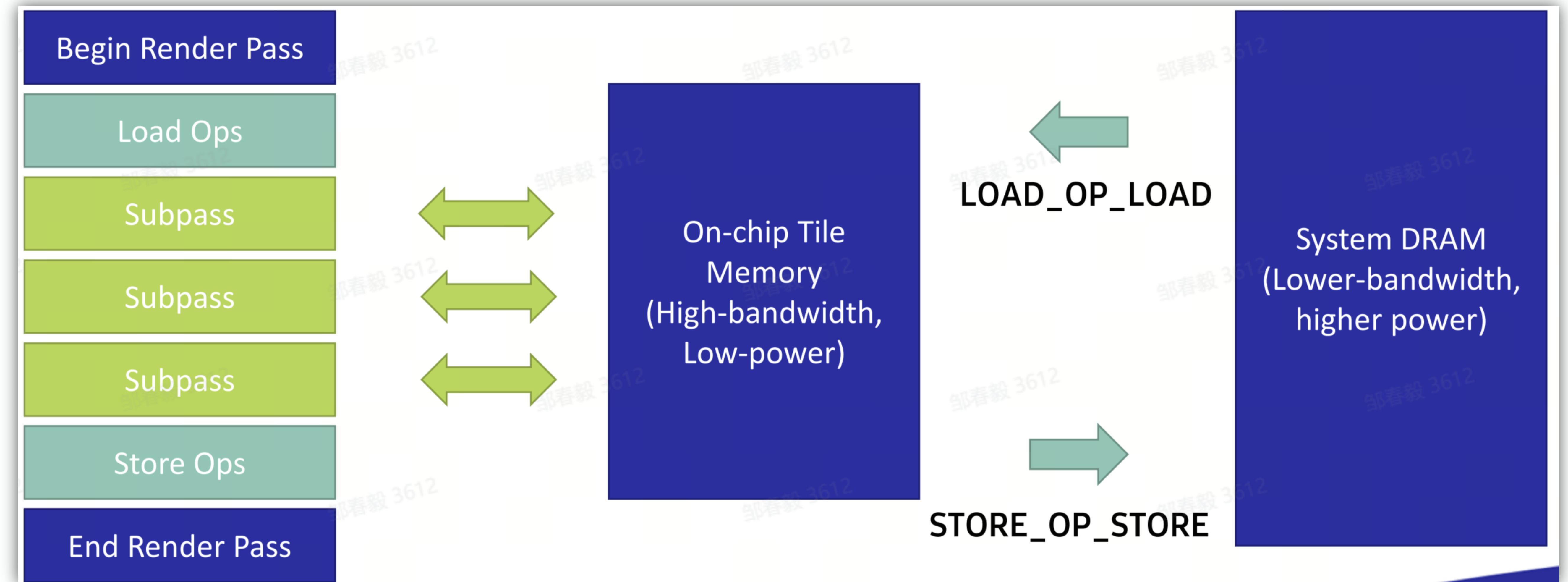
PICO视频

ByteDance字节跳动

PICO视频

颜色空间转换

Vulkan上基于SubPass的解决方案

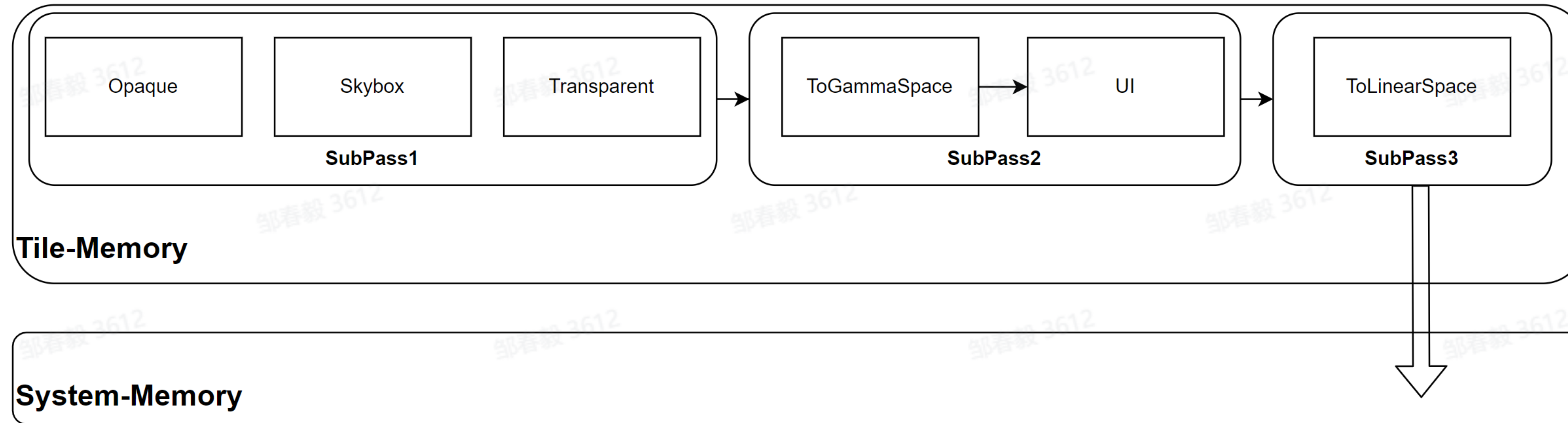


- FrameBufferFetch隐形的实现当前像素的可读写，SubPass需要显性的控制当前像素的可读写
- Render1->Render2->Render3...RenderN，FrameBufferFetch可以在每一个Render渲染时获得其之前的FrameBuffer数值，即Render3可以获得Render2的渲染结果
- SubPass1(Render1->Render2)->SubPass2(Render3->Render4->Render5...RenderN)，每一个SubPass只能获得其之前SubPass渲染结束后的FrameBuffer数值，即Render5只能获得Render2渲染后的结果
- 2022年的新增扩展**VK_EXT_rasterization_order_attachment_access**可以实现GL_EXT_shader_framebuffer_fetch的等级效果 https://github.com/KhronosGroup/Vulkan-Docs/blob/main/proposals/VK_EXT_rasterization_order_attachment_access.adoc#resolved-what-are-the-differences-to-the-vk_arm_rasterization_order_attachment_access-extension

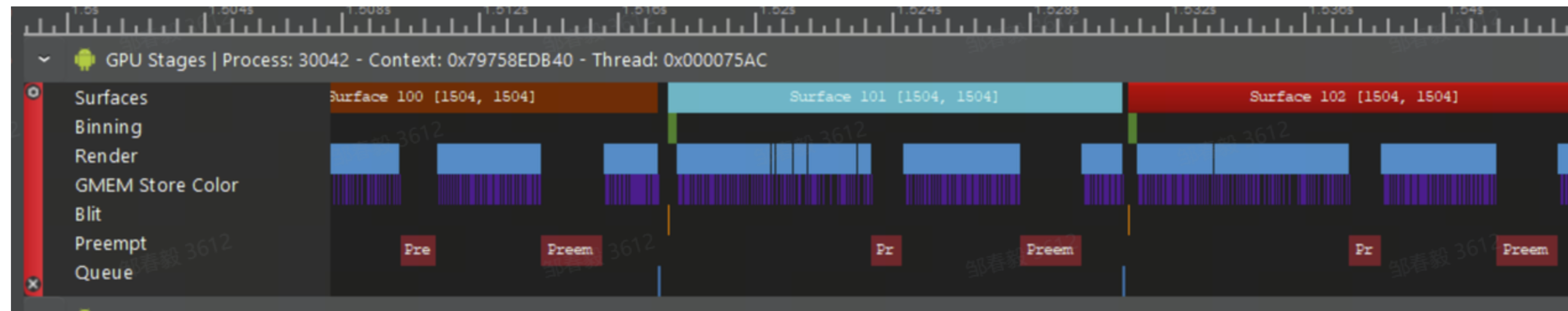
ByteDance字节跳动

颜色空间转换

Vulkan上基于SubPass的解决方案



上述为基于SubPass的渲染方式，SubPass1作为SubPass2的输入，SubPass2作为SubPass3的输入，进而实现传统需要切换RenderTarget才能实现的效果，期间只在最后SubPass渲染结束后才写回主存

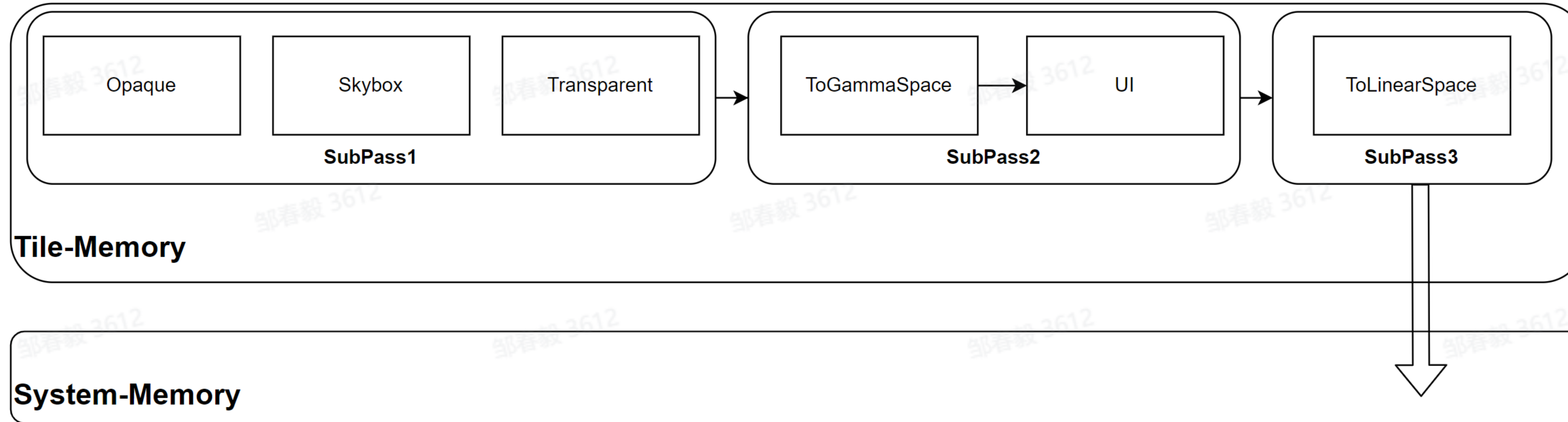


PICO 视频

ByteDance字节跳动

颜色空间转换

SubPass与MSAA的兼容性问题



SubPass1提供给SubPass2的InputAttachment需要手动处理MSAA

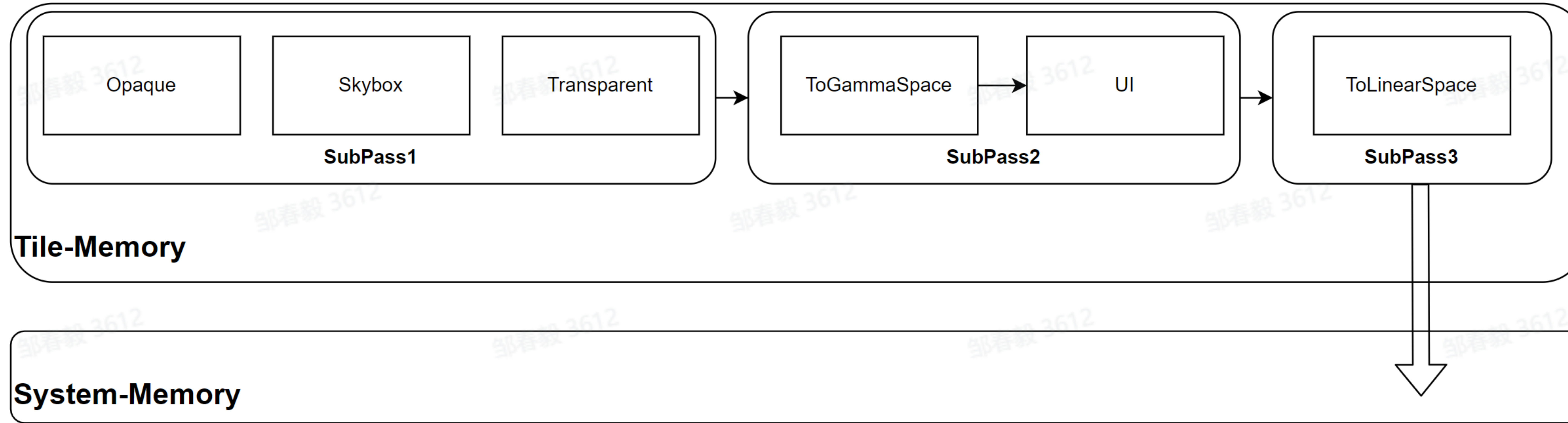
```
#if _SubPassMSAA2 || _SubPassMSAA4
UNITY_DECLARE_FRAMEBUFFER_INPUT_FLOAT_MS(0);
#else  #if _SubPassMSAA2 || _SubPassMSAA4
UNITY_DECLARE_FRAMEBUFFER_INPUT_FLOAT(0);
#endif  #if _SubPassMSAA2 || _SubPassMSAA4 #else
```

```
#if _SubPassMSAA2
float4 col = UNITY_READ_FRAMEBUFFER_INPUT_MS(0, 0, input.positionCS) * 0.5 + UNITY_READ_FRAMEBUFFER_INPUT_MS(0, 1, input.positionCS) * 0.5;
#elif _SubPassMSAA4  #if _SubPassMSAA2
float4 col = 0;
[unroll(4)]
for (int i = 0; i < 4; ++i)
{
    col += UNITY_READ_FRAMEBUFFER_INPUT_MS(0, i, input.positionCS) * 0.25;
}
#else  #elif _SubPassMSAA4
float4 col = UNITY_READ_FRAMEBUFFER_INPUT(0, input.positionCS);
#endif  #elif _SubPassMSAA4 #else
```

PICO 视频
ByteDance字节跳动

颜色空间转换

SubPass与MSAA的兼容性问题



SubPass3保存到System-Memory的内容支持自动Resolved

```
if (backColorDescriptor.msaaSamples > 1)
{
    final.ConfigureResolveTarget(new RenderTargetIdentifier(backColorTexture.Identifier(), mipLevel: 0, CubemapFace.Unknown, depthSlice: -1));
    final.loadStoreTarget =
        new RenderTargetIdentifier(BuiltinRenderTextureType.None, mipLevel: 0, CubemapFace.Unknown, depthSlice: -1);
}
else
{
    final.ConfigureTarget(new RenderTargetIdentifier(backColorTexture.Identifier(), mipLevel: 0, CubemapFace.Unknown, depthSlice: -1), loadExistingContents: false, storeResults: true);
}
```

PICO 视频
ByteDance字节跳动

VRS(可变速率着色)

VRS

什么是**VRS(Variable Rate Shading/**可变速率着色**)**

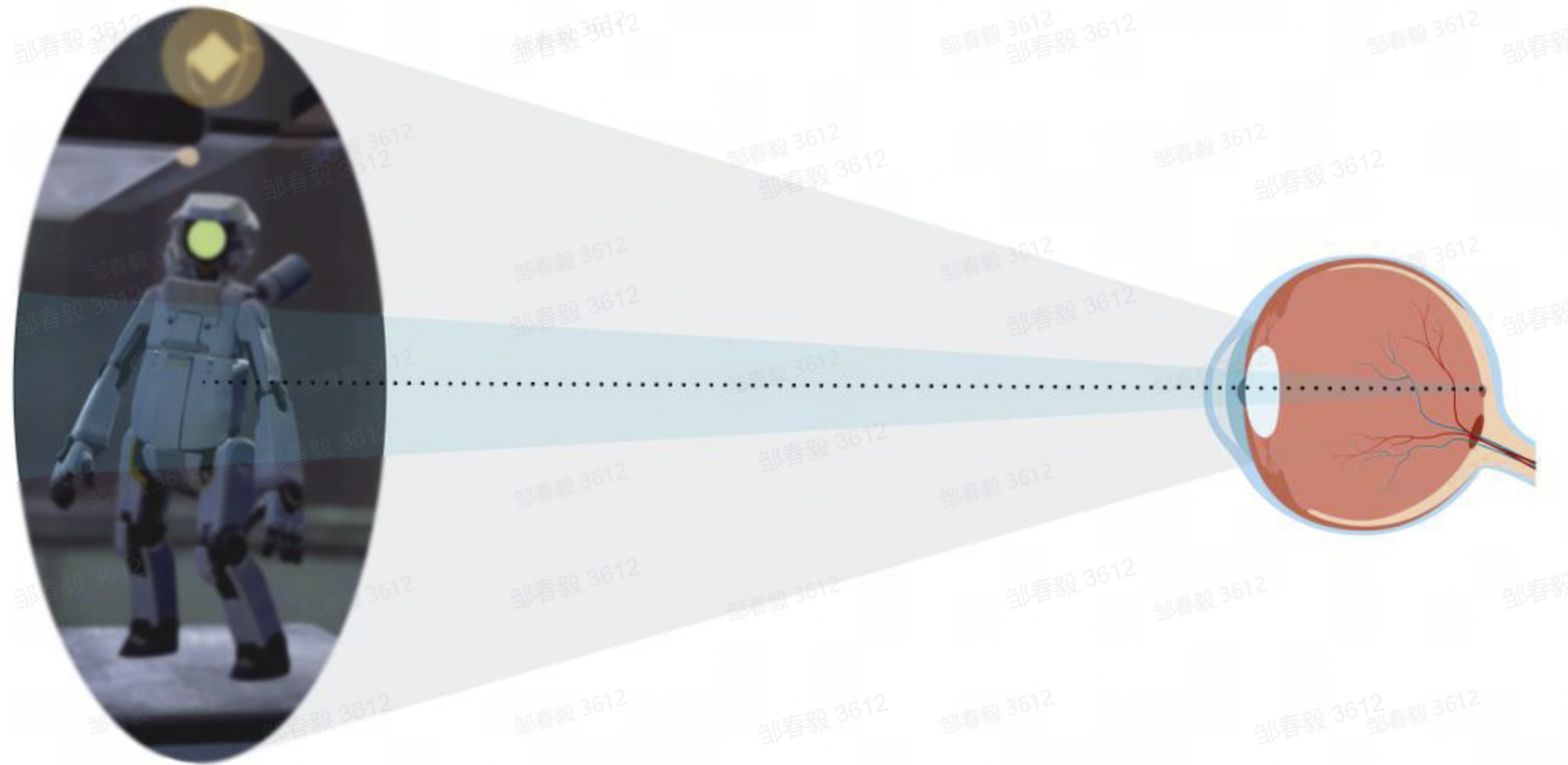
正常情况下，一张RenderTexture的像素密度是相同的，针对每一块区域需要渲染的像素也是相同的。如果将整张RT不同区域的每2x2的区域只渲染1个像素，这样这张RT就会产生不同的像素密度，这就是可变速率着色。

https://developer.nvidia.com/vrworks/graphics/variablerateshading

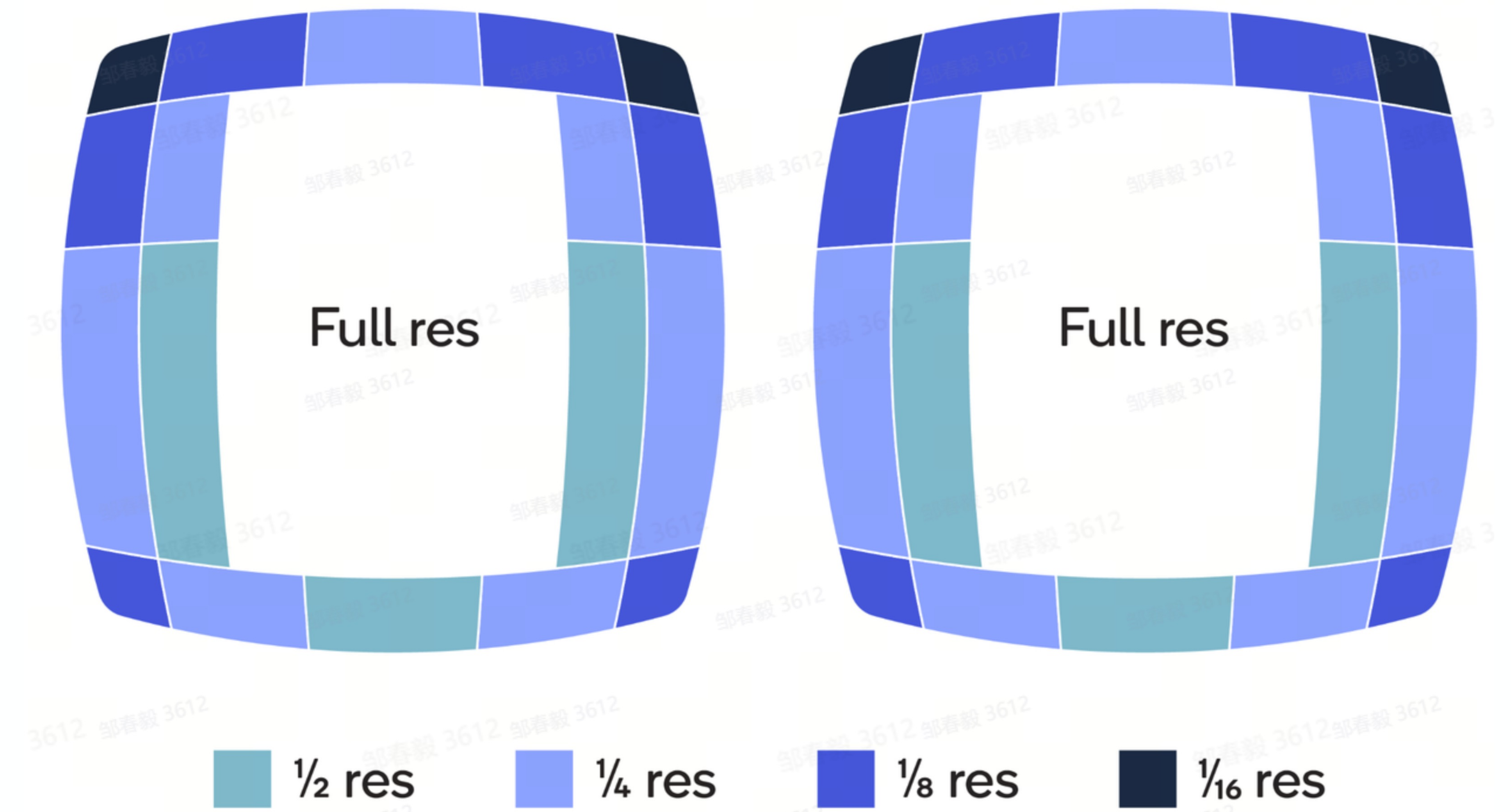
VRS的核心要点是有选择的控制RT不同区域的像素密度，进而降低渲染的性能开销。

注视点渲染

什么是注视点渲染(Foveated Rendering)



人眼在观察事物会呈现视野聚焦区域清晰，周围模糊的特点，因此借助这一特点可以让人眼注视的区域采用高分辨率，周围的区域采用低分辨率，进而在不影响观感的情况下降低性能的消耗。

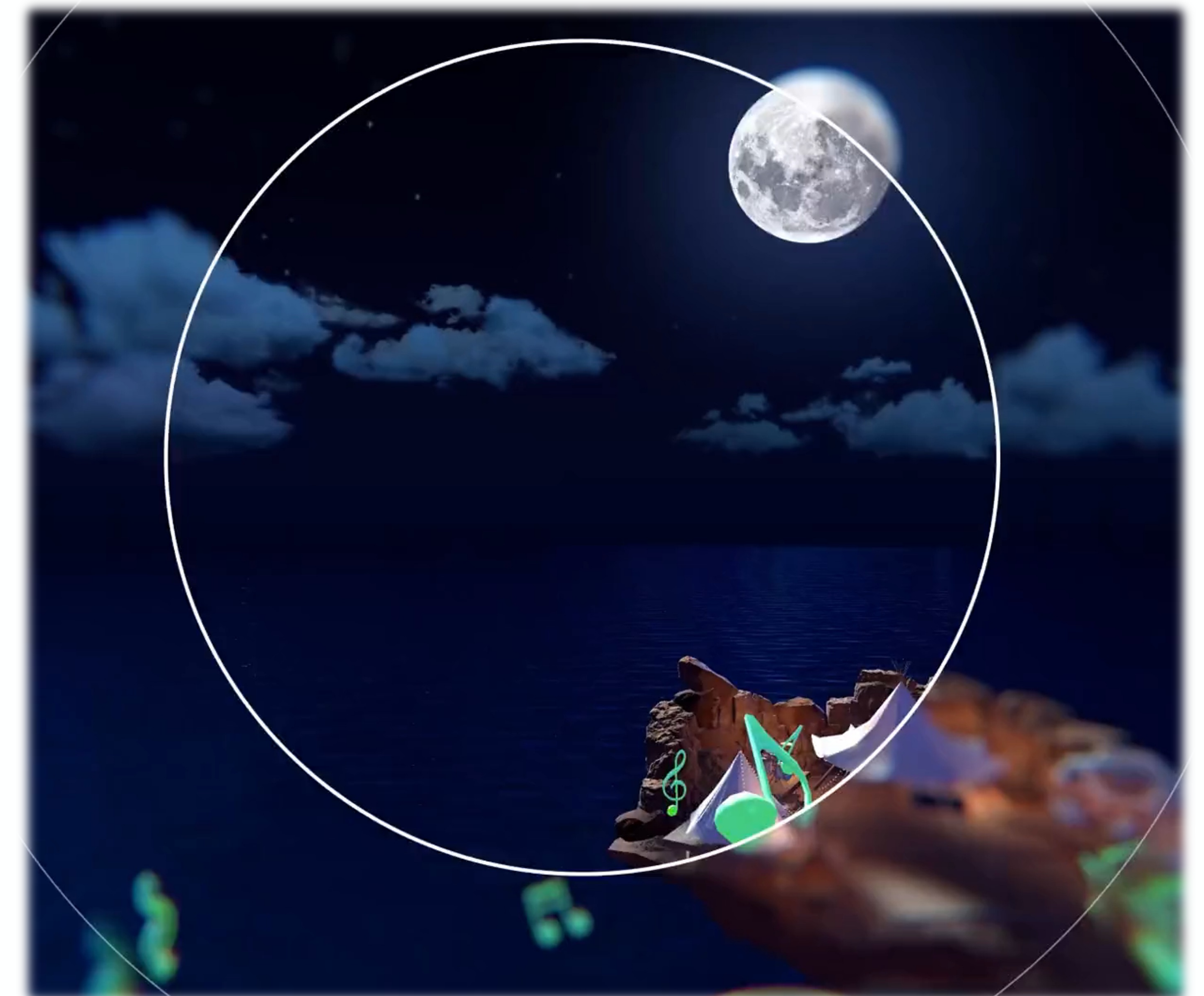


如果固定高分辨率区域在屏幕的中心区域，这种方案称之为静态注视点渲染，即FFR。

注视点渲染

什么是注视点渲染(Foveated Rendering)

如果借助眼动追踪硬件设备，进而让高分辨区域可以始终保持在眼球在屏幕的聚焦点上，这种方案称之为眼动追踪注视点渲染，即ETFR。



PICO视频

注视点渲染

注视点渲染的实现(OpenGLES版本)

QCOM_texture_foveated

https://registry.khronos.org/OpenGL/extensions/QCOM/QCOM_texture_foveated.txt

C#端为CommandBuffer扩展注释点渲染的2个接口

```
        bool foveatedRenderingEnabled = false;
        if (TryGetFoveatedRenderingParameters(out TextureFoveatedParameters param))
        {
#if PICO_VIDEO_VRS_EXTEND2_SUPPORTED
            cmd.EnableTexFoveatedFeature(textureID, param.foveationMinimum, subsampledLayoutEnabled);
#else
            cmd.EnableTexFoveatedFeature(textureID, param.foveationMinimum);
#endif
            cmd.SetTexFoveatedParameters(textureID, focalPoint: 0, focalPointX: s_FoveatedRenderingFocalPoint.x, focalPointY: s_FoveatedRenderingFocalPoint.y, param.foveationGainX, param.foveationGainY, param.foveationArea);
            foveatedRenderingEnabled = true;
        }
```

C++端通过CommandBuffer为Texture绑定扩展

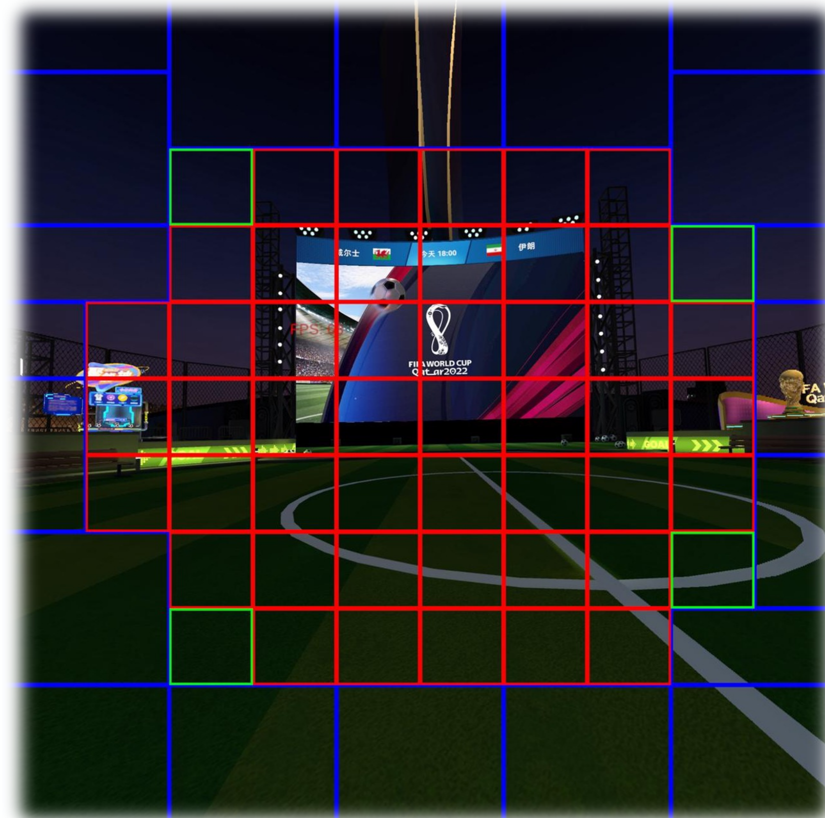
```
inline void SetTexFoveatedFeature(RenderingCommandBuffer *self, int nameID , float foveationMinimum, bool subsampledLayout)
{
    ShaderLab::FastPropertyName rtName; rtName.index = nameID;
    self->SetTexFoveatedFeature(rtName, 0x1 | (subsampledLayout ? 0x4 : 0x2), foveationMinimum);
}

inline void SetTexFoveatedParameters(RenderingCommandBuffer *self,
    int nameID,
    int focalPoint,                         // focalPoint
    float focalPointX,                      // focalX
    float focalPointY,                      // focalY
    float foveationGainX,                   // gainX
    float foveationGainY,                   // gainY
    float foveationArea)
{
    ShaderLab::FastPropertyName rtName; rtName.index = nameID;
    self->SetTexFoveatedParameters(rtName, focalPoint, focalPointX, focalPointY, foveationGainX , foveationGainY, foveationArea);
}
```

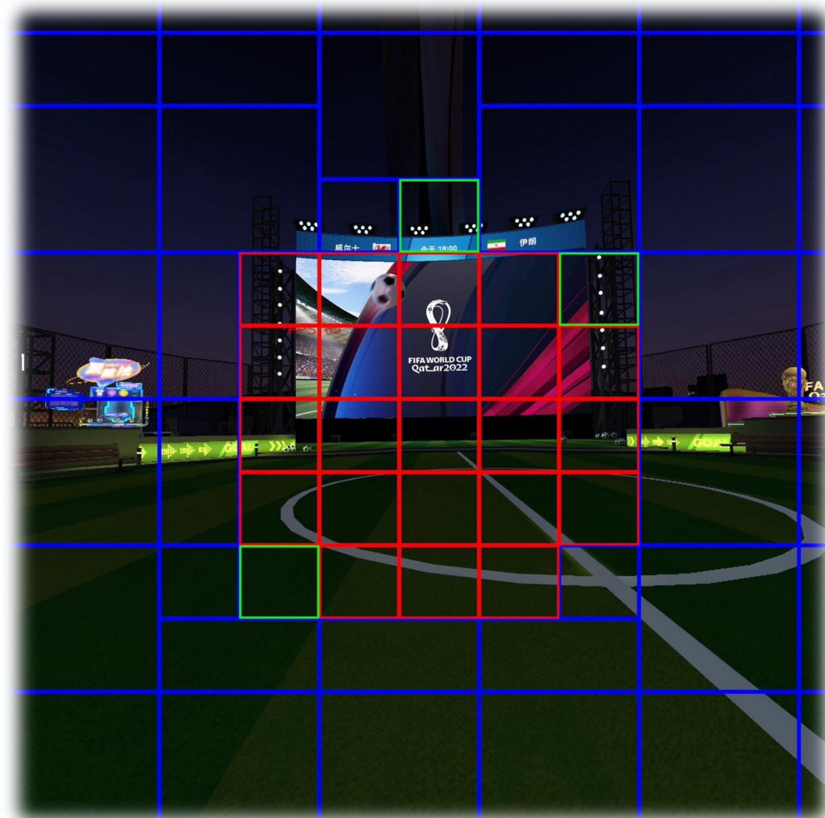
```
void ApiGLES::SetTexFoveatedFeature(GLESTexture* tex, unsigned int feature, float foveationMinimum) const
{
#if PLATFORM_ANDROID
    GLES_CALL(this, glBindTexture, tex->target, tex->texture);
    GLES_CALL(this, glTexParameteri, tex->target, GL_TEXTURE_FOVEATED_FEATURE_BITS_QCOM, feature);
    GLES_CALL(this, glTexParameterf, tex->target, GL_TEXTURE_FOVEATED_MIN_PIXEL_DENSITY_QCOM, foveationMinimum);
    GLES_CALL(this, glBindTexture, tex->target, 0);
#endif
}

void ApiGLES::SetTexFoveatedParameters(
    GLESTexture* tex,
    int focalPoint,                         // focalPoint
    float focalPointX,                      // focalX
    float focalPointY,                      // focalY
    float foveationGainX,                   // gainX
    float foveationGainY,                   // gainY
    float foveationArea                     // foveaArea
) const
{
#if PLATFORM_ANDROID
    for (size_t i = 0; i < tex->layers; i++)
    {
        GLES_CALL(this, glTextureFoveationParametersQCOM, tex->texture, i, focalPoint, focalPointX, focalPointY, foveationGainX, foveationGainY, foveationArea);
    }
#endif
}
```

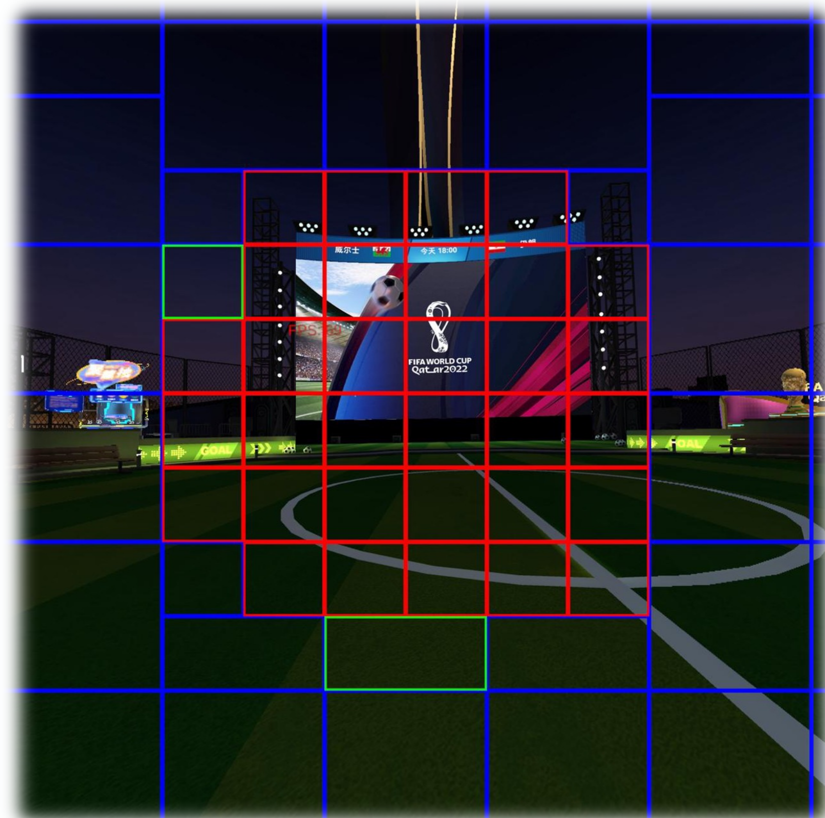
ByteDance字节跳动

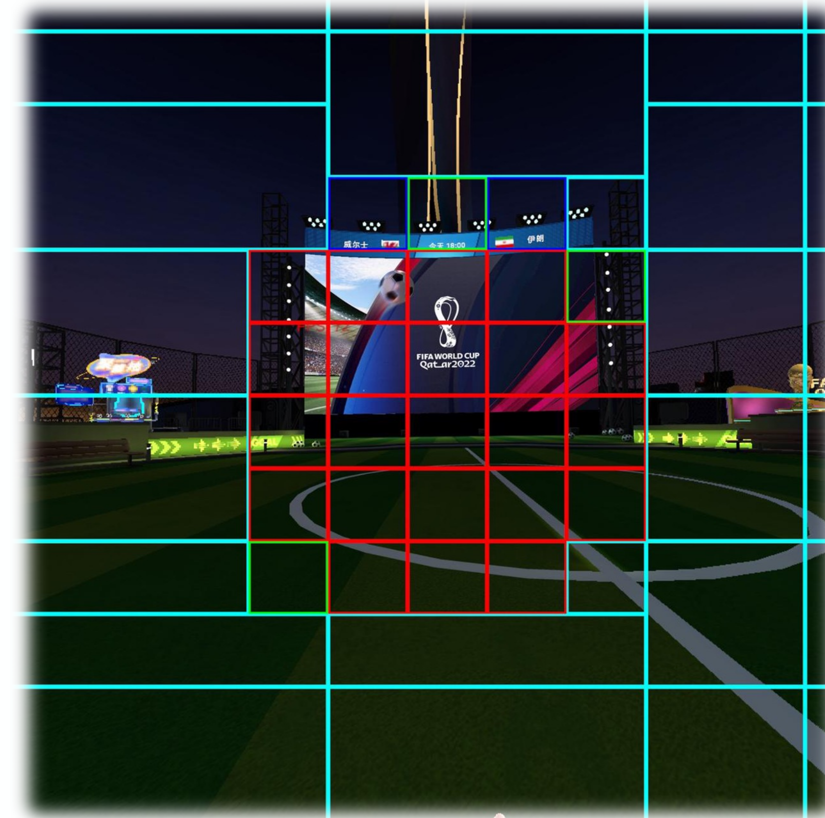
注视点渲染

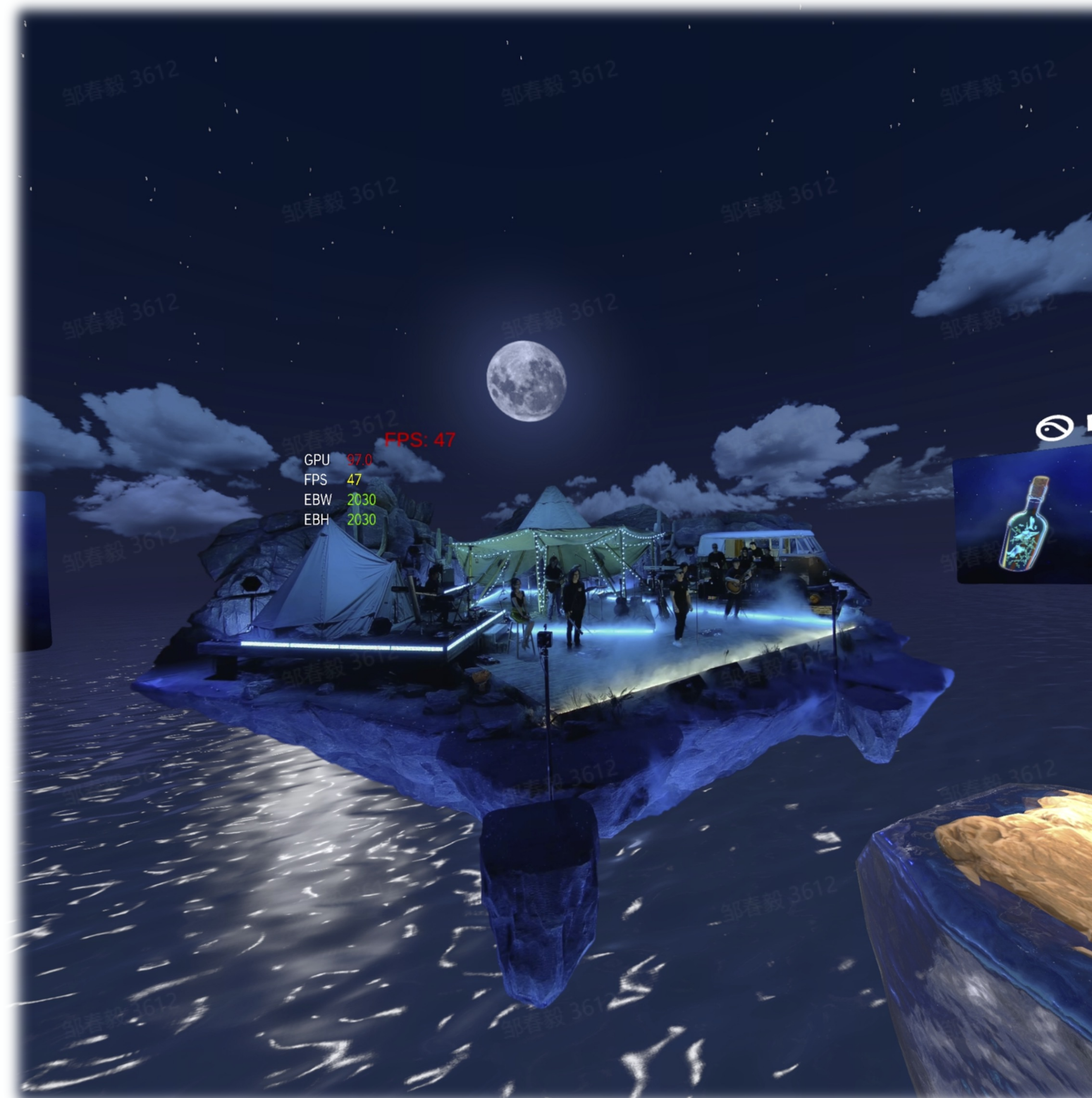
FFR不同级别的密度图


Low

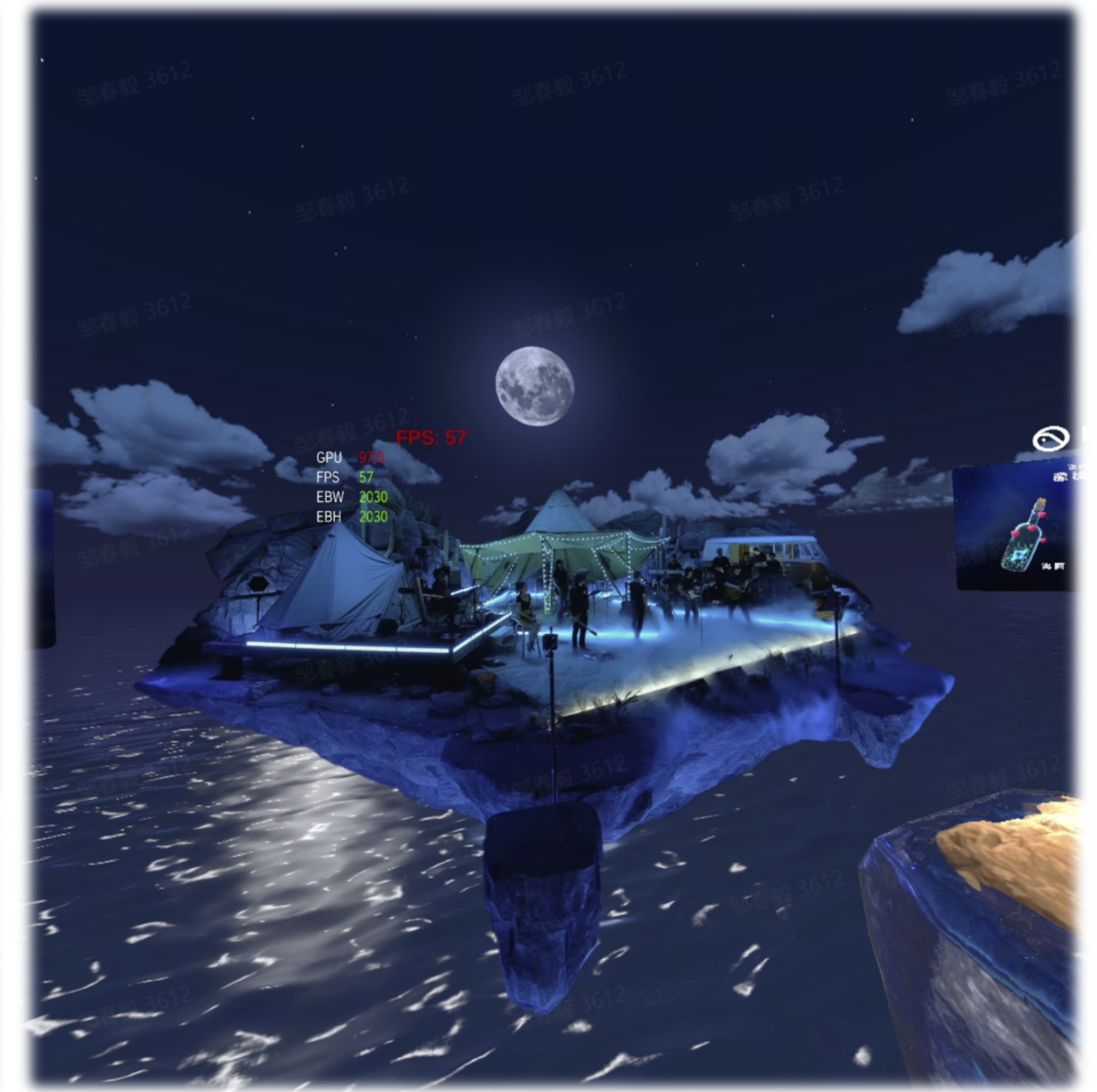

High


Medium


TopHigh



关闭FFR



开启FFR(Medium)

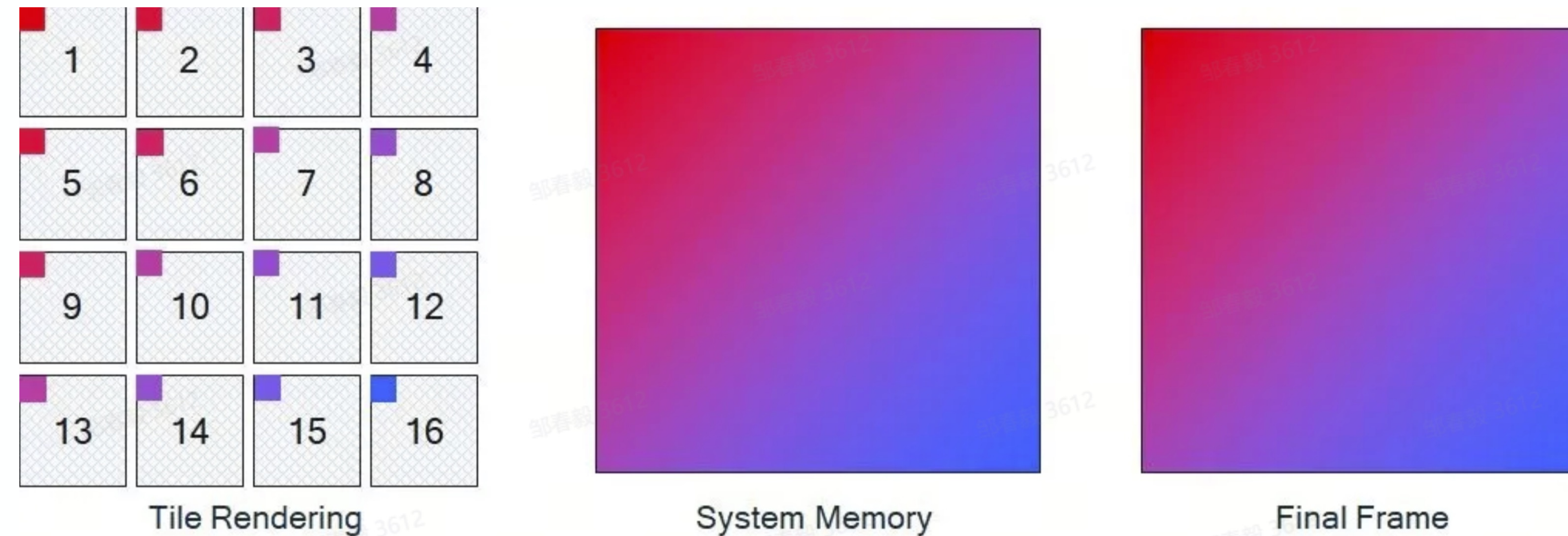
PICO视频

注视点渲染

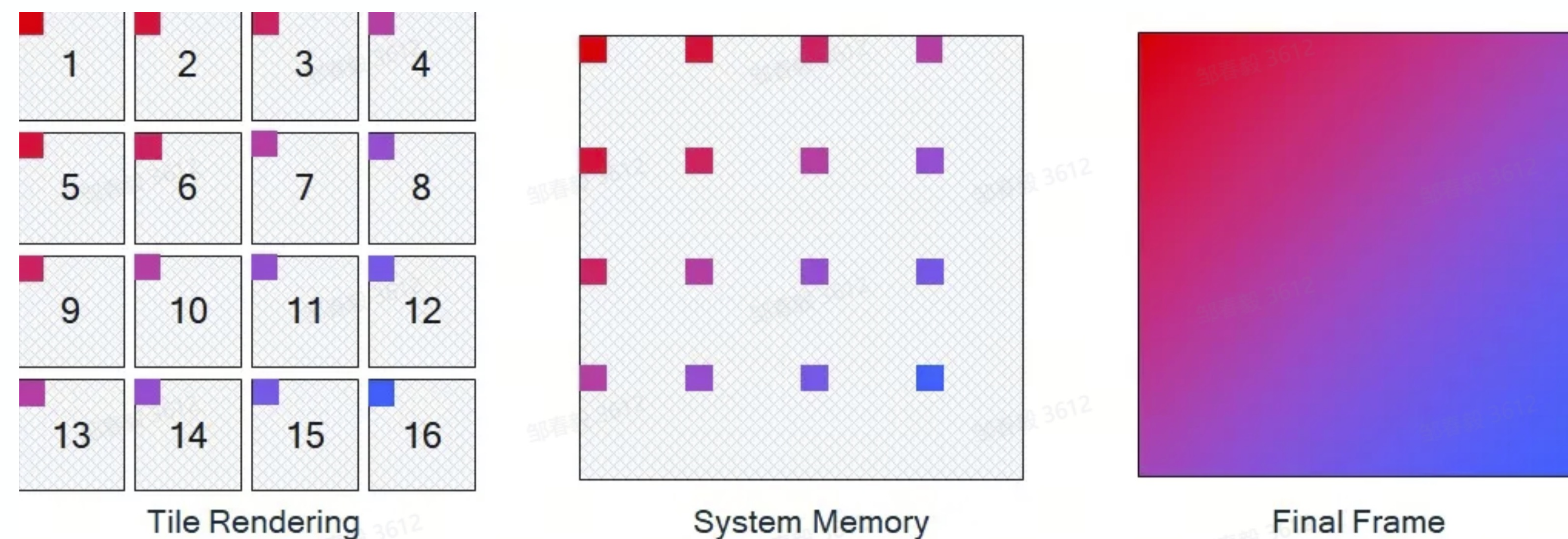
SubsampledLayout

https://registry.khronos.org/OpenGL/extensions/QCOM/QCOM_texture_foveated_subsampled_layout.txt

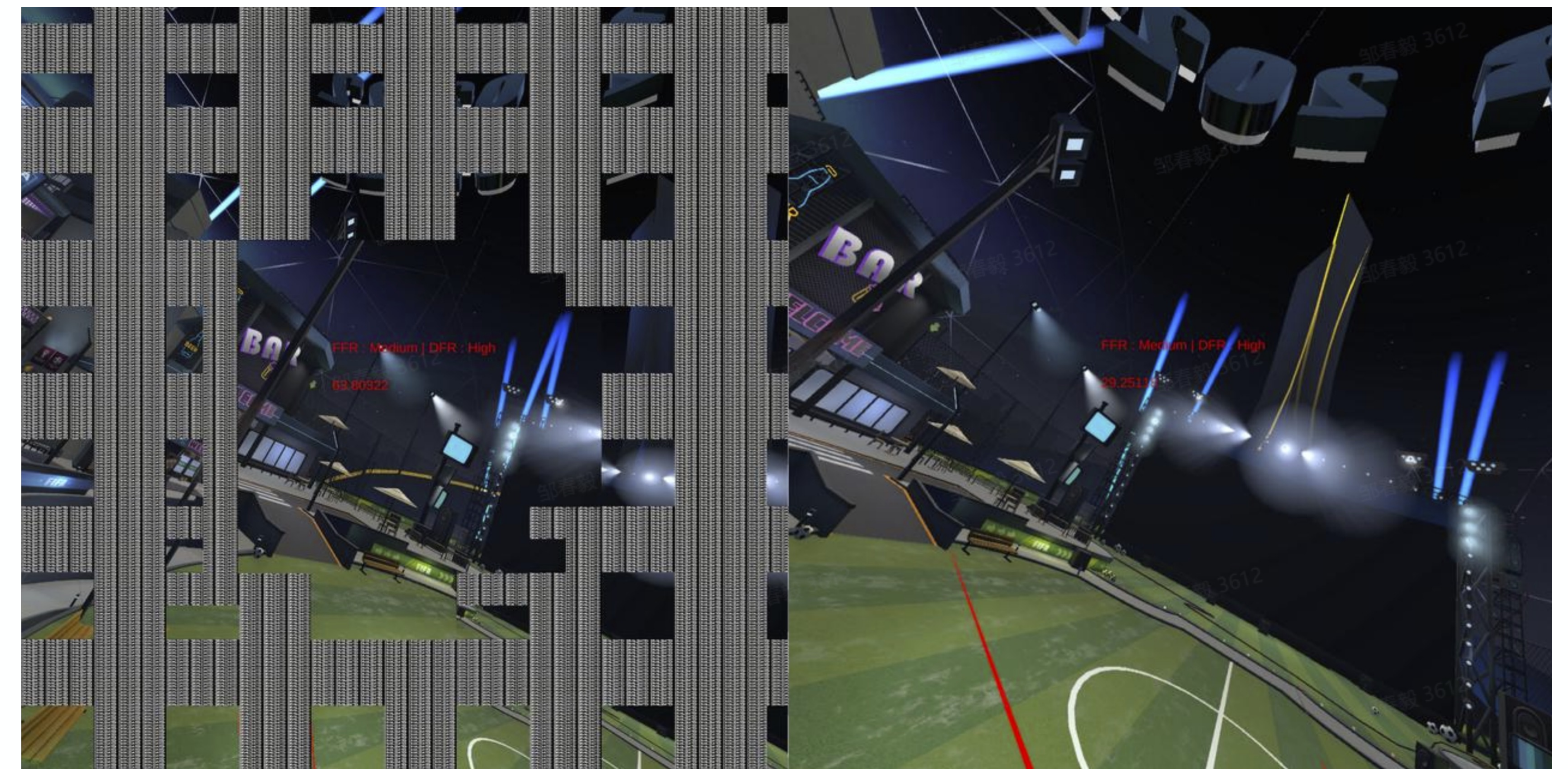
Subsampled Layout是基于Foveated Rendering的一个高级扩展。开启Subsampled Layout的情况下，在Tile Rendering之后System Memory只会存储低分辨率的贴图，在进行最终采样时才会通过Upscaling还原到原始分辨率。相比较原始Foveated Rendering可以减少带宽消耗以及一定程度上降低边缘降分辨率带来的渲染混叠现象。



标准foveated rendering,system memory存储原始大小的数据



启用subsampled layout,system memory存储降分辨率后的数据，在最终采样时还原



ByteDance字节跳动

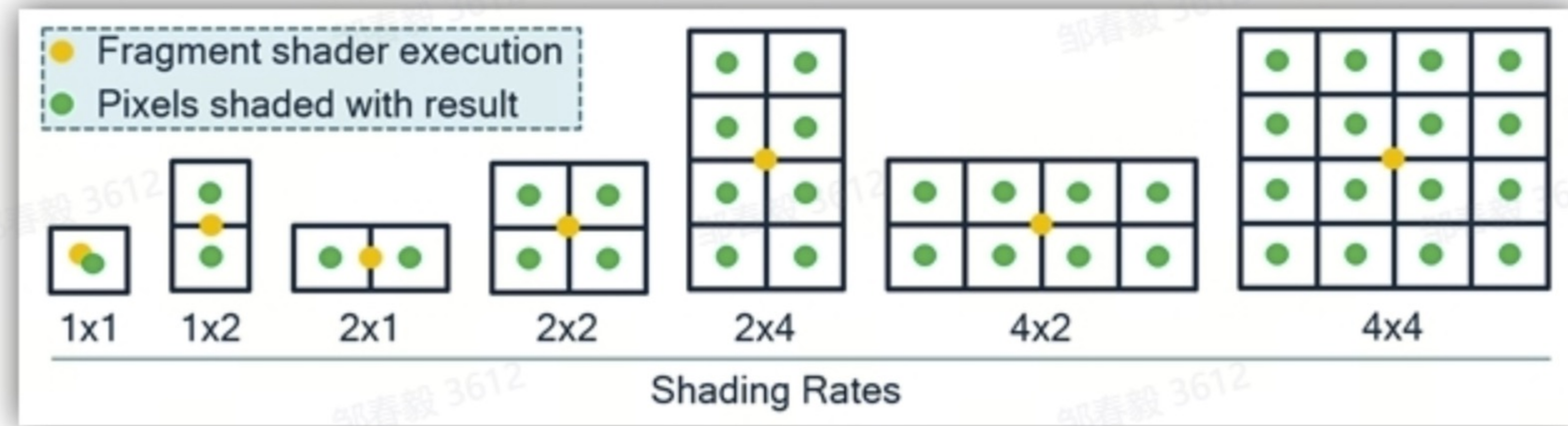
注视点渲染

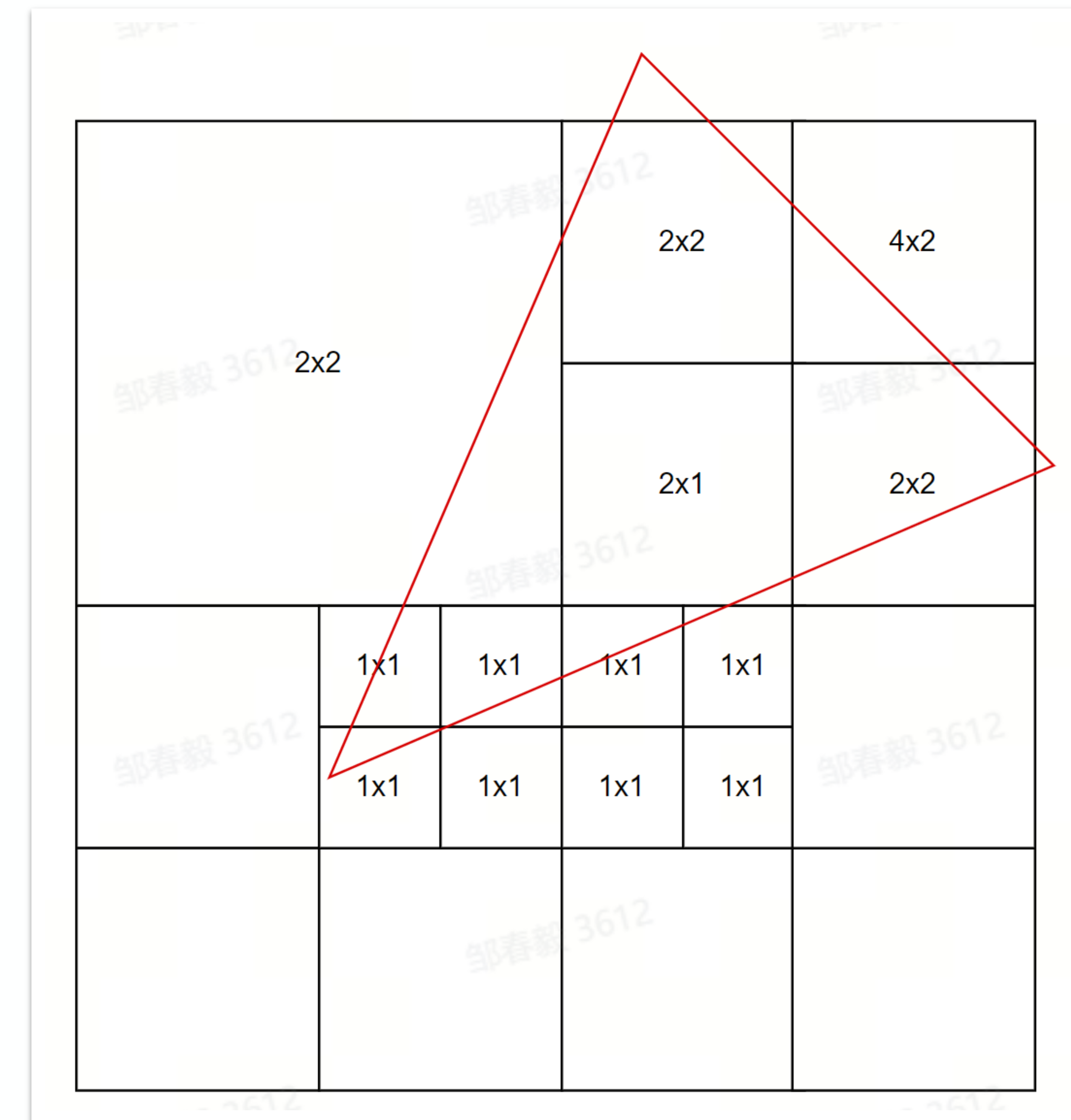
ShadingRate扩展

https://registry.khronos.org/OpenGL/extensions/QCOM/QCOM_shading_rate.txt

相较于FR的降采样率模式，QCOM_shading_rate扩展允许针对某一次CommandBuffer提交整体进行降采样进而提升渲染效率。

```
cmd.SetShadingRate(Shad);
if (m_ActiveCame   ShadingRateLevel.ShadingRate1x1    ShadingRateLevel
{                  ShadingRateLevel.ShadingRate1x2    ShadingRateLevel
    //PicoVideo;   ShadingRateLevel.ShadingRate2x1    ShadingRateLevel
    UniversalRen   ShadingRateLevel.ShadingRate2x2    ShadingRateLev
IDEO_RT_FORCE_DEST ShadingRateLevel.ShadingRate4x2    ShadingRateLev
                   ShadingRateLevel.ShadingRate4x4    ShadingRateLev
```





注视点渲染

性能测试数据

- FR的收益主要在于像素着色的压力，这种主要来源于计算时芯片的算力，另一方面来源于数据传输的带宽。
- 像素着色压力=分辨率x像素着色复杂度
- 整个场景的像素着色压力越大，那么FR的收益就会越高，因为本次测试采用高分辨率来增加像素着色压力，进而测试各个FR功能的性能优化潜质。

注视点渲染

性能测试数据

PICO 视频

ByteDance字节跳动

注视点渲染

性能测试数据

THANKS