Advanced Shading Techniques with Pixel Local Storage

Roberto Lopez Mendez November 12, 2021

13 minute read time.

In part one of this blog series, we provided a general overview of the Pixel Local Storage (PLS) extension from today's perspective and when it was

first launched in 2014. Part two takes a more in-depth look at the advanced shading techniques that are possible through PLS.

Translucency

Since the moment the development of PLS was publicly shared at SIGGRAPH 2013, its importance for implementing advanced rendering and post-processing effects on mobile was highlighted. Before this, translucency-like effects required several passes with the consequent memory flush out and data read back from main memory. Bandwidth consumption was a practical impediment for many complex rendering techniques on mobile.

Let's have a look at how such an advanced effect can be implemented using PLS. Translucency is the effect of light passing slightly through a material, something in between fully opaque and fully transparent. The wax in candles and tree leaves are common examples of translucent materials. To realistically render these materials, we need to consider Subsurface Scattering (SSS). This is the light transport mechanism that explains how the light that penetrates the surface of a translucent object is scattered by interacting with the material and exiting the surface at a different point. We all easily recognize this effect, for example, when light passes through ears.

A rough implementation of SSS on mobile will consider how light is attenuated as it travels through a material. The attenuation can be influenced by several factors. These include the varying thickness of the object, the view direction, and the properties of the light. In practice, we need to determine how far the light travels inside an object before it

reaches the point to be shaded. A fair approximation is to compute the object thickness as seen from the camera (t), instead of the actual distance travelled by the light (s).

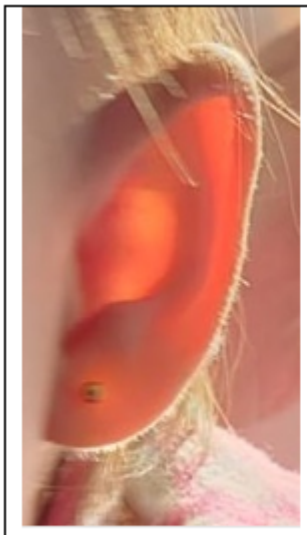


Figure 1. Translucency effect in the ear.

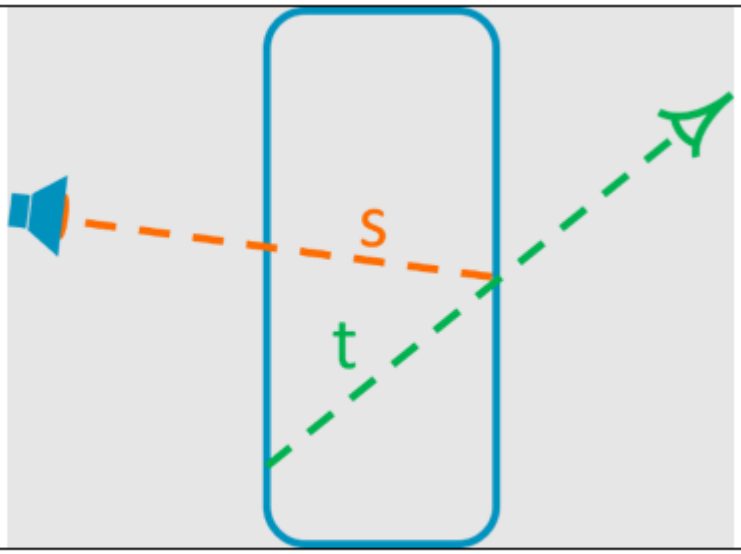


Figure 2. Approximate light attenuation considering thickness as viewed from the camera.

Following this approach, we will compute the thickness as the difference between the maximum and the minimum view-space depth of an object. We will do this in two passes.

The first pass determines the closest object and its minimum view-space depth. Opaque and translucent objects are differentiated by writing an ID value of zero and >= 1 respectively in the stencil buffer. The below PLS block is allocated. Please note that the total size of the structure is 128 bits.

Fullscreen

```
__pixel_localEXT FragDataLocal {
    layout(rgb10_a2) vec4 lighting; // RGBA
    layout(rg16f) vec2 minMaxDepth; // View-space depths
    layout(rgb10_a2) vec4 albedo;   // RGB and sign(normal.z)
    layout(rg16f) vec2 normalXY;    // View-space normal components
} storage;
```

In this first pass, the fragment shader writes the material properties and sets both minimum and maximum depth to be the incoming depth. The lighting variable is cleared as it will be used to accumulate lighting in the second pass.

Fullscreen

```
uniform vec3 albedo;
in vec4 vClipPos;
in vec4 vPosition;
in vec3 vNormal;

void main()
{
    vec3 n = normalize(vNormal);
    storage.lighting = vec4(0.0);
    storage.minMaxDepth = vec2(-vPosition.z, -vPosition.z);
    storage.albedo.rgb = albedo;
    storage.albedo.a = sign(n.z);
    storage.normalXY = n.xy;
}
```

In the second pass, the same PLS block is used, and the shader finds the maximum depth of the previously determined closest object. This time the scene (see Fig. 3) is rendered without depth testing, but with a stencil test set to equal each object's ID. Since the ID of the closest object is stored in the stencil buffer, only that same object will have fragments that pass through.

Fullscreen

```
void main()
{
    float depth = -vPosition.z;
    storage.minMaxDepth.y = max(depth, storage.minMaxDepth.y);
}
```

The picture below on the left shows the rendering of the resulting thickness of the translucent objects.

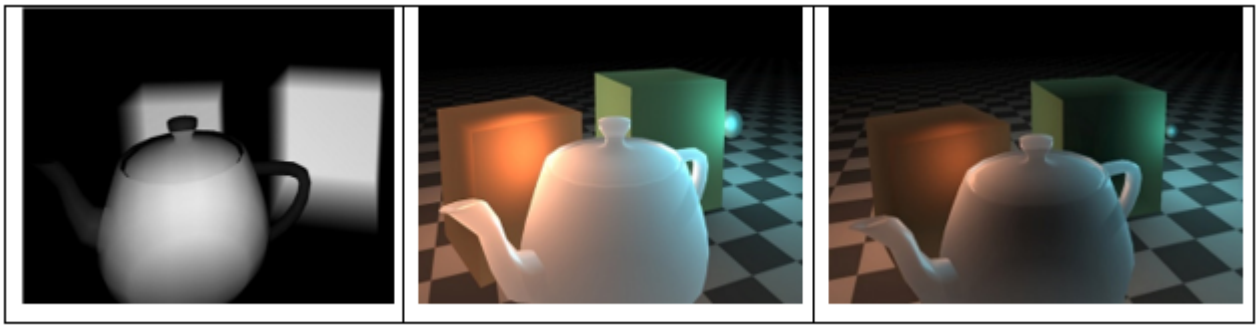


Figure 3. Left: Thickness rendering. Center: Translucency rendering. Right: Lower-intensity light rendering.

Once the material properties and the object view-space thickness have been computed and stored in the PLS block, a last shading pass over all translucent geometry is performed (central picture). The basic idea of this pass is that the thickness attenuates light transmittance. The larger the thickness, the smaller the transmitted light intensity will be. For very thin objects, we should see a large intensity. A detailed explanation of this pass and the source code can be found in the Arm OpenGL ES SDK for Android. In the translucency example, two lights move in the opposite direction, both going through the cubes. You can play with the translucency example by changing some parameters and see the resulting effect (e.g. picture on the right).

Order Independent Transparency (OIT)

Usually, any geometry with some transparency is rendered using alpha compositing. Each semi-transparent geometry occludes the preceding one and adds some of its own color depending on its alpha value. The order in which geometry is blended is relevant. For example, Arm Mali GPU Best Practices Guide recommends to first render all opaque meshes in a front-to-back render order and then, to ensure blending works correctly, render all transparent meshes in a back-to-front render order, over the top of the opaque geometry.

Depending on the number and complexity of the semi-transparent geometry, the ordering can take a significant amount of time and not always produce the right result. Alternative approaches known as Order Independent Transparency (IOT) have been implemented. OIT sorts geometry per-pixel, after rasterization. An exact solution that accurately computes the final color would require all fragments to be sorted and this could become a bottleneck for complex scenes. More mobile-friendly approximate solutions offer a balance between quality, performance, and memory bandwidth. Among these we can mention Multi-Layer Alpha Blending (MLAB), Adaptive Transparency and Depth Peeling.

MLAB is a real-time approximated OIT solution that operates in a single rendering pass and in bounded memory. In other words, a perfect fit for PLS. It works by storing transmittance and depth, together with accumulated color in a fixed number of layers. New fragments are inserted in order, when possible, and merge, when required. This way the strict ordering requirements of alpha blending can be somewhat relaxed. For the simplest case of two layers, the blended layers can be accumulated in the PLS structure and resolve them to a single output color at the end. A presentation at SIGGRAPH 2014 compares different approaches for OIT and recommends that the PLS implementation of MLAB uses RGBA8 for color and alpha, and 32-bit float for depth.

Frag0 Colour	Frag1 Depth	Frag1 Colour	Frag1 Depth
RGBA8	R32F	RGBA8	R32F

Table 1. PLS structure for Multi-Layer Alpha Blending (2 layers) implementation.

The depth value is used to determine whether a first fragment 0 is behind or in front of a second fragment 1. The over/under operator is then applied accordingly. The pseudo-code below shows the operations to be

implemented in the shader. Figure 8 shows the resulting compositing image when using the MLAB approach described here for two layers, and the reference image. Although this is the simplest MLAB approach and it improves on common alpha blending, we observe an obvious artefact on the lower left sphere.

Fullscreen

```
If(frag0Depth < frag1Depth)
{
    //Fragment 0 covers fragment 1
    accumColorRGB = frag0ColorRGB + frag1ColorRGB * (1 – frag0Alpha)
    AccumAlpha = frag0Alpha + frag1Alpha *(1 - frag0Alpha)
}
else
{
    //Fragment 1 covers fragment 0
    accumColorRGB = frag1ColorRGB + frag0ColorRGB * (1 – frag1Alpha)
    AccumAlpha = frag1Alpha + frag0Alpha *(1 – frag1Alpha)
}
```

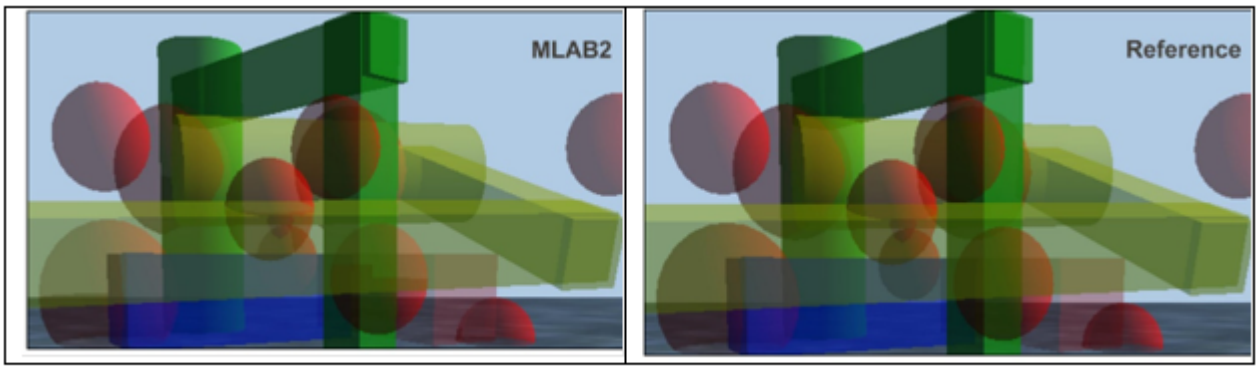


Figure 4. Left: OIT image composition of two layers using the MLAB approach. Right: The reference compositing image of the two layers generated using depth peeling method.

Shader Frame Buffer Fetch (FBF)

Before finishing this blog, I would like to highlight another couple of extensions that expose Mali fast on-chip memory to developers, and thus offer a substantial bandwidth saving alternative. The first extension, Arm Frame Buffer Fetch, provides fragment shaders read access to the previous fragment color. It enables fragment shaders to read existing framebuffer

data as input and is relevant to implement use-cases such as programmable blending, and any other operations that may not be possible to implement with fixed-function blending. This extension is supported in OpenGL ES 2.0 or higher and provides single framebuffer readback, but with a more efficient approximate MSAA path than EXT_shader_framebuffer_fetch.

The use of this extension is straightforward as it works through the new built-in gl_LastFragColorARM (mediump vec4). The example below showcases programmable blending.

Fullscreen

```
#extension GL_ARM_shader_framebuffer_fetch : enable
precision mediump float;
uniform vec4 uBlend0;
uniform vec4 uBlend1;

void main(void)
{
    vec4 color = gl_LastFragColorARM;
    color = lerp(color, uBlend0, color.w * uBlend0.w);
    color *= uBlend1;
    gl_FragColor = color;
}
```

The second extension is Arm Frame Buffer Fetch Depth and Stencil, which allows fragment shaders to read the current depth and stencil values from the framebuffer. The usage is also straightforward through two new read-only built-ins: lowp int gl_LastFragStencilARM for accessing the stencil buffer and gl_LastFragDepthARM for accessing the depth buffer. For depth, the precision depends on whether GL_FRAGMENT_PRECISION_HIGH is defined (highp float) or not (mediump float). These variables contain, respectively, the current value of the depth and stencil buffer for the pixel to which the current fragment is destined. This extension enables use cases such as programmable depth and stencil testing, modulating shadows, soft particles and the ability to create variance shadow maps in a single render pass. This extension also offers applications a very convenient method of reconstructing 3D

positions of any pixel on the screen using depth and camera matrix. The example below[1] showcases the use of depth read-back to render soft particles. Normally, this would require two render passes: one that writes out the depth values of the background geometry to a depth texture, and one that renders the particles while reading from the depth texture to do the blending. This extension allows all this to be done in a single pass.

Fullscreen

```
#extension GL_ARM_shader_framebuffer_fetch_depth_stencil : enable
precision mediump float;

uniform float uTwoXNear;        // 2.0 * near
uniform float uFarPlusNear;     // far + near
uniform float uFarMinusNear;    // far - near
uniform float uInvParticleSize;
uniform sampler2D uParticleTexture;

varying float vLinearDepth;
varying vec2 vTexCoord;

void main(void)
{
    vec4 ParticleColor = texture2D(uParticleTexture, vTexCoord);
    //Convert from exponential depth to linear
    float LinearDepth = uTwoXNear / (uFarPlusNear - gl_LastFragDepthAR
                        uFarMinusNear);

    //Compute blend weight by substracting current fragment depth with
```

Feedback

The extension specification states that reads from gl_LastFragDepthARM and gl_LastFragStencilARM need to wait for the processing of all previous fragments destined for the current pixel to complete. To achieve the best performance, it is therefore recommended that reads from either of these built-in variables are done as late in the execution of the fragment shader as possible. Please read the extension documentation for some other specific recommendations, for example when multi-sampling is enabled.

Things to be aware of

As we have seen, PLS and FBF extensions provide a number of advantages to OpenGL ES developers. Nevertheless, there are some things we need to be aware of when planning to use these extensions. The first one is that not all vendors support PLS. In fact, this extension is currently only supported by Arm and Imagination, so you will find it available in Arm Mali and Imagination PowerVR GPUs.

This partial support across vendors makes the adoption of PLS harder for those wishing to target all of Android with a single code base. This kind of challenge was one of the reasons why the graphics community started looking for a new API that all vendors could support, with this leading to Vulkan. Vulkan's render passes provided an opportunity to express similar semantics to PLS, but in a cross-platform manner – albeit in a more limited and more implicit form. In Vulkan, when a render pass contains multiple subpasses, which use the subpass load functionality to pass per-pixel data between the passes, the GPU has enough information to keep this data in on-chip storage and avoid the store/load round trip to memory. This is referred to as 'subpass fusion' and all Mali Vulkan drivers support this. However, whether the GPU uses DRAM or on-chip storage is not explicitly controlled by the application, so this behavior does vary between vendors. Nevertheless, as it stands, in Vulkan we cannot directly access tile storage like in the OpenGL ES extensions.

Additionally, the use of the on-chip memory in both OpenGL ES and Vulkan is based on the premise that each pixel must only use the information from the corresponding input pixel. Neither of them allows developers to sample from an arbitrary pixel location, much less the previous content. This would open the door for implementing highly optimized new effects on mobile.

Lastly, although all graphics APIs are designed to make the most of the underlying hardware, they are not a guarantee for implementing high performance low power mobile apps. There is no substitute for profiling

and testing. Nothing but patient profiling work will tell us whether the GPU stalls or is thirsty for commands, and thus undoes any advantage taken from the tile memory extensions. This is what will ultimately tell us where the bottleneck is and help us make decisions for an optimal use of runtime resources. For Arm Mali GPUs, Arm Mobile Studio provides access to several profiling and monitoring tools such as Streamline Performance Analyzer, Performance Counters, Performance Advisor and Mali Offline Compiler. These resources combined with following the Mali GPU Best Practices are the truly most powerful tools that graphics developers can use to make the most of Arm Mali GPUs.

Conclusions

I hope that at this point an OpenGL ES developer reader can have an idea of how to use the PLS and FBF extensions to implement efficient graphics on mobile. Nevertheless, when going through the different examples discussed in the two blogs, it is important to understand the main concept behind PLS. For the first time in graphics, PLS exposed to developers the on-chip storage and with this the possibility of building a shading pipeline using persistent memory. Suddenly, a number of use-cases that were out of the reach of mobile became possible, mainly thanks to the drastic reduction of bandwidth. For mobile platforms, power is a precious resource. By allowing storing and reading data on the on-chip memory, less power is used moving data to and from the main memory. This means that the battery lasts longer. Finally, by keeping data in the on-chip memory, we drastically reduce the runtime resources used on data traffic, increasing performance. When planning your next application using OpenGL ES, keep in mind that you get all these benefits for free if you use PLS.

Annex

The table below intends to provide a guide to developers about when to use the extensions described in the blogs. It summarizes the use-cases covered by the extensions and highlights useful features.

Extension	OpenGL ES version	Use when	Relevant feature
EXT_shader_pixel_local_storage	3.0 or higher	Your application can pass information between fragment shaders invocations covering the same pixel. Ex. Deferred rendering, translucency, OIT.	The data in PLS is not written back to main memory. Massive bandwidth saving.
ARM_shader_framebuffer_fetch	2.0 or higher	You have a programmable blending like use case. Fixed-function blending doesn't support the operation you need.	Enables fragment shaders to read existing framebuffer colour data as input at pixel or fragment location. No bandwidth cost associated since the data is in the tile buffer.
ARM_shader_framebuffer_fetch_depth_stencil	2.0 or higher	Your app needs to read existing framebuffer depth and stencil of the current pixel being processed in raster order access.	Enables fragment shaders to read existing framebuffer depth/stencil data as input at pixel or fragment location. No bandwidth cost associated since the data is in the tile buffer.

References

[1] - Bandwidth Efficient Graphics with Arm Mali GPUs, Marius Bjorge, GPU Pro 5, p. 275.

Graphics, Gaming, and VR blog

Performance analysis with Arm Mobile Studio

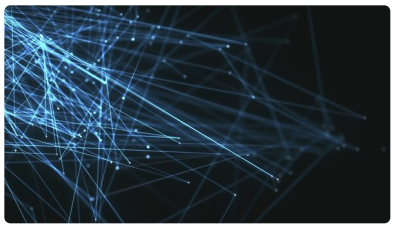
In part 3 of Arm's Mali GPU training series, learn how to analyze the performance of a mobile game with Arm Mobile Studio, our free-to-use performance analysis tool suite.

Julie Gaskin

Best practice principles for mobile game development

Part 2 of Arm's free Mali GPU training for mobile graphics developers. Here, we present the latest best practice recommendations to get the best from devices with Mali GPUs.

Julie Gaskin



Best practices to resolve typical multi-context rendering issues

Read the best practices for application that have multi context render operation with Mali CSF based GPUs.

Fred.Li