Arm® Mali™ GPU OpenGL ES 3.x

Version 1.1

Developer Guide

Non-Confidential

Issue 01
100587_0101_01_en

Arm® Mali™ GPU OpenGL ES 3.x
Developer Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	11 March 2016	Non-Confidential	Initial release
0101-01	14 January 2022	Non-Confidential	First release of version 1.1

Proprietary Notice

Confidentiality Status

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Signal names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace bold`	Language keywords when used outside example code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
⚠ Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
Note	An important piece of information that needs your attention.
Tip	A useful tip that might make it easier, better or faster to perform a task.
Remember	A reminder of something important that relates to the information you are reading.

1.2 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-2: Arm publications

Document Name	Document ID	Licensee only
None	-	-

Note

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 Introduction to OpenGL ES 3.x on Mali GPUs

This chapter introduces the ARM Mali GPU OpenGL ES 3.x Developer Guide.

2.1 Additional Reading

The following table provides links to third-party websites that you can visit.

Table 2-1: Additional Reading

Title	Author	Web Link
Polygonise	Paul Bourke	http://paulbourke.net/geometry/polygonise
Dist Functions	Iquilezeles	http://iquilezles.org/www/articles/distfunctions/distfunctions.htm
Geometry	Open.gl	https://open.gl/geometry
Smooth Voxel Terrain (Part2)	0fps.net	http://0fps.net/2012/07/12/smooth-voxel-terrain-part-2

2.2 About OpenGL ES 3.0

OpenGL ES 3.0 is an enhancement to OpenGL ES 2.0 standard.

OpenGL ES 3.0 adds features that are already present in OpenGL 3.x. The additional OpenGL ES 3.0 features include:

- Enhancements to the OpenGL ES rendering pipeline that enable acceleration of advanced visual effects. These include:
    - Occlusion queries.
    - Transform feedback.
    - Instanced rendering.
    - Support for four or more rendering targets.
- High-quality texture compression as a standard feature. This means you can use the same textures on different platforms.
- A new version of the GLSL ES shading language with full support for integer and 32-bit floating point operations.

- Enhanced texturing functionality that includes:
    - Guaranteed support for floating point textures.
    - 3D textures.
    - Depth textures.
    - Vertex textures.
    - NPOT textures.R/RG textures.
    - Immutable textures.
    - 2D array textures.
    - Swizzles.
    - LOD and mip level clamps.
    - Seamless cube maps.
    - Sampler objects.
- A set of specifically sized texture and render-buffer formats that are guaranteed to be present. This helps you write portable applications because the variability between implementations is reduced.

Note

OpenGL ES 3.0 is backwards compatible with OpenGL ES 2.0.

2.3 About OpenGL ES 3.1

OpenGL ES 3.1 is an enhancement to the existing OpenGL ES 3.0 standard.

OpenGL ES 3.1 adds features that are already present in OpenGL 4.x. The additional OpenGL ES 3.1 features include:

- Compute shaders.
- Indirect draw calls.
- Memory resources:
    - Images.
    - Shader Storage Buffer Objects.
- Enhanced texturing:
    - Texture gather.
    - Multisample textures.
    - Stencil textures.
- Separate shader objects.

- Shading language features:
    - Arrays of arrays.
    - Bitfield operations.
    - Location qualifiers.
    - Memory read and write.
    - Synchronization primitives.

Note

OpenGL ES 3.1 is backwards compatible with OpenGL ES 2.0 and 3.0.

2.4 About the Android Extension Pack

The Android Extension Pack adds additional features to OpenGL ES 3.1.

AEP is a set of OpenGL ES extensions that bring console-class gaming to Android. AEP features include:

- ASTC (LDR) texture compression format.
- Compute shaders.
- Geometry shaders.
- Tessellation.
- Per-sample interpolation and shading.
- Different blend modes for each color attachment in a frame buffer.
- Guaranteed fragment shader support for shader storage buffers, images, and atomics.

Note

AEP is an optional feature in Android L on platforms that support OpenGL ES 3.1. AEP is included in the OpenGL ES 3.2 standard.

2.5 About OpenGL ES 3.2

OpenGL ES 3.2 adds the Android Extension Pack and additional functionality into the core OpenGL ES standard. It enables desktop-class graphics in mobile, consumer, and automotive hardware. OpenGL ES 3.2 features include:

- Geometry and tessellation shaders.
- Floating point render targets.

- ASTC texture compression.
- Enhanced blending.
- Advanced texture targets:
    - Texture buffers.
    - Multisample 2D arrays.
    - Cube map arrays.
- Debug and robustness features.
- OpenGL ES 3.2 is backwards compatible with OpenGL ES 2.0, 3.0 and 3.1.

3 The Mali GPU Hardware Families

This section describes the different Mali GPU architecture families and their hardware features.

3.1 About the Mali GPU hardware families

The current families of Mali GPUs available are:

- Midgard,
- Bifrost,
- Valhall.

The most up-to-date list of processor families can be found on our Developer website: Arm GPU Architecture.

The latest GPU datasheet can also be found on Developer as a PDF: Arm Mali GPU Datasheet PDF.

3.2 About the Mali GPU hardware features

Mali GPUs contain the following common hardware:

3.2.1 Tile-based deferred rendering

The Mali GPU divides the framebuffer into tiles and renders it tile by tile. Tile-based rendering is efficient because values for pixels are computed using on-chip memory. This technique is ideal for mobile devices because it requires less memory bandwidth and less power than traditional rendering techniques.

3.2.2 L2 cache controller

One or more L2 cache controllers are included with the Mali GPUs. L2 caches reduce memory bandwidth usage and power consumption.

An L2 cache is designed to hide the cost of accessing memory. Main memory is typically slower than the GPU, so the L2 cache can increase performance considerably in some applications.

Note

Mali GPUs use L2 cache in place of local memory.

3.3 Mali GPU shader cores

The shader cores handle the vertex and fragment processing stages of the graphics pipeline.

The shader cores generate lists of primitives and accelerate the building of data structures, such as polygon lists and packed vertex data, for fragment processing.

The shader cores also handle the rasterization and fragment processing stages of the graphics pipeline. They use the data structures and lists of primitives that are generated during vertex processing to produce the framebuffer result that appears on the screen.

3.3.1 Bifrost architecture hardware components

The Bifrost family of Mali GPUs use the same top-level architecture as the earlier Midgard GPUs. They use a unified shader core architecture. This means that the design only includes a single type of hardware shader processor.

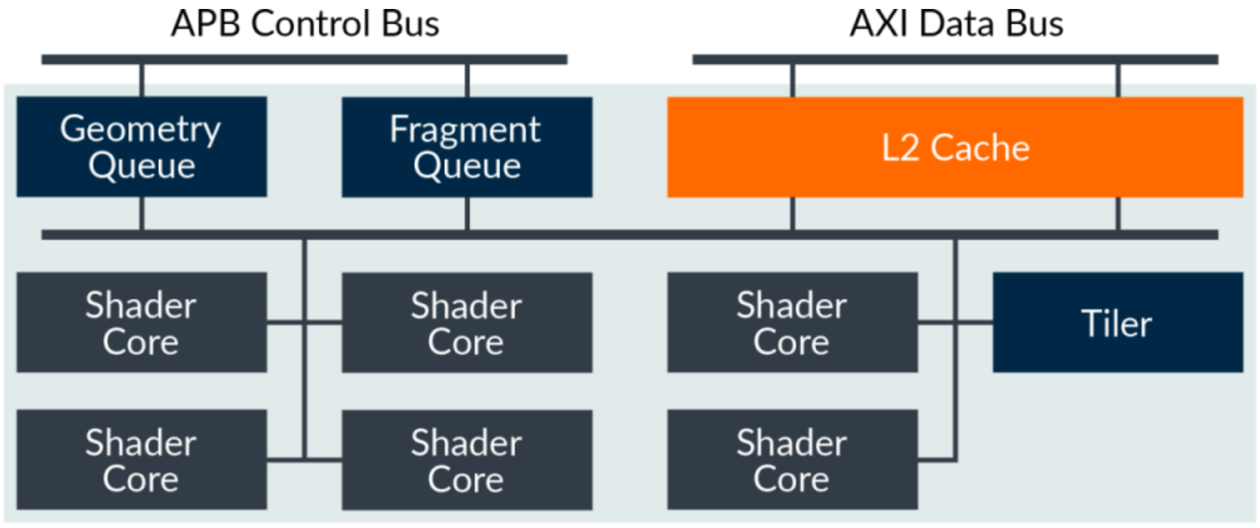
This single shader processor can execute all types of shader code, such as vertex shaders, fragment shaders, and compute kernels.

The exact number of shader cores present in a silicon chip can vary. We license configurable designs to our silicon partners, who can then choose how to configure the GPU in their specific chipset, based on their performance needs and silicon area constraints.

For example, the Mali-G72 GPU can scale from a single core, for low-end devices, up to 32 cores for the highest performance designs.

The following diagram provides a top-level overview of the Control Bus and Data Bus of a typical Mali Bifrost GPU:

Figure 3-1: Mali Bifrost GPU.



To improve performance, and to reduce memory bandwidth wastage caused by repeated data fetches, the shader cores in the system all share access to a level 2 cache. The size of the L2 cache, while configurable by our silicon partners, is typically in the range of 64-128KB per shader core in the GPU. However, the size of the L2 cache depends on how much silicon area is available.

Also, our silicon partners can configure the number, and bus width, of the memory ports that the L2 cache has to external memory.

The Bifrost architecture aims to write one 32-bit pixel, per core, per clock. Therefore, it is reasonable to expect an eight-core design to have a total of 256-bits of memory bandwidth, for both read and write, per clock cycle. This can vary between chipset implementations.

3.3.2 Work issue

Once the application has completed defining the render pass, the Mali driver submits a pair of independent workloads for each render pass.

One independent pass deals with all geometry and compute related workloads. The other independent pass is for the fragment-related workload. As Mali GPUs are tile-based renderers, all geometry processing for a render pass must be complete before the fragment shading can begin.

A finalized tile rendering list provides the fragment processing pass with the per-tile primitive coverage information that it needs.

Bifrost GPUs can support two parallel issue queues. One for geometry/compute workloads, and the other for fragment workloads. This arrangement allows the workload to be distributed across all available shader cores in the GPU.

3.4 Valhall architecture hardware components

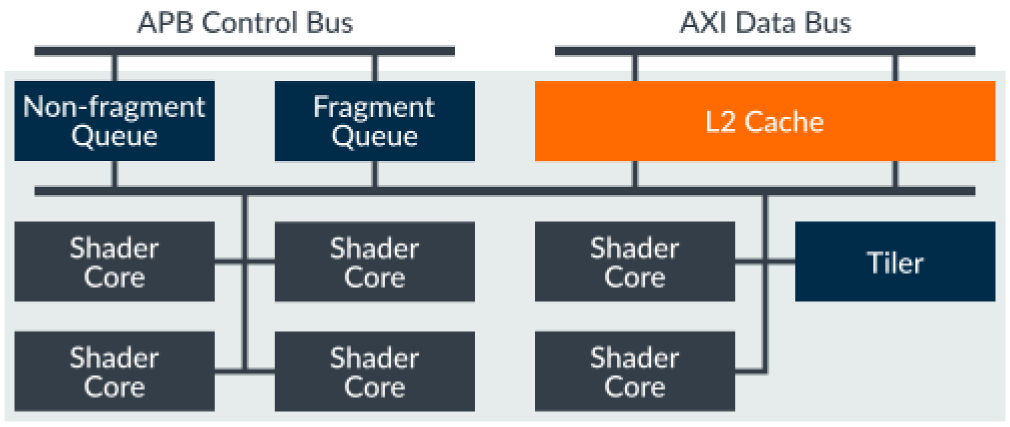
The Valhall family of Mali GPUs uses the same top-level architecture as the previous generation Bifrost GPUs. The Valhall family uses a unified shader core architecture.

This means that only a single type of hardware shader processor exists in the design. This single processor type can execute all types of shader code, including vertex shaders, fragment shaders, and compute kernels.

The exact number of shader cores that are present in a silicon chip can vary. At Arm we license configurable designs to our silicon partners. Partners choose how to configure the GPU in their chipset based on their performance needs and silicon area constraints.

The following diagram provides a top-level overview of the control bus and data bus of a typical Mali Valhall GPU:

Figure 3-2: Mali Valhall GPU.



To improve performance, and to reduce memory bandwidth wastage that is caused by repeated data fetches, the shader cores in the system all share access to a level 2 cache.

The size of the L2 cache is configurable by our silicon partners and typically is in the range of 64-128KB for each shader core in the GPU. The size of the L2 cache depends on how much silicon area is available.

Also, our silicon partners can configure the number, and bus width, of the memory ports that the L2 cache has to external memory.

The Valhall architecture aims to write two 32-bit pixels per core per clock. Therefore, it is reasonable to expect a 4-core design to have a total of 256 bits of memory bandwidth, for both read and write, per clock cycle. This can vary between chipset implementations.

4 The OpenGL ES graphics pipeline

Mali GPUs implement a graphics pipeline supporting the OpenGL ES *Application Programming Interfaces* (APIs).

4.1 About the OpenGL ES 3.x pipeline

Mali GPUs use data structures and hardware functional blocks to implement the OpenGL ES graphics pipeline.

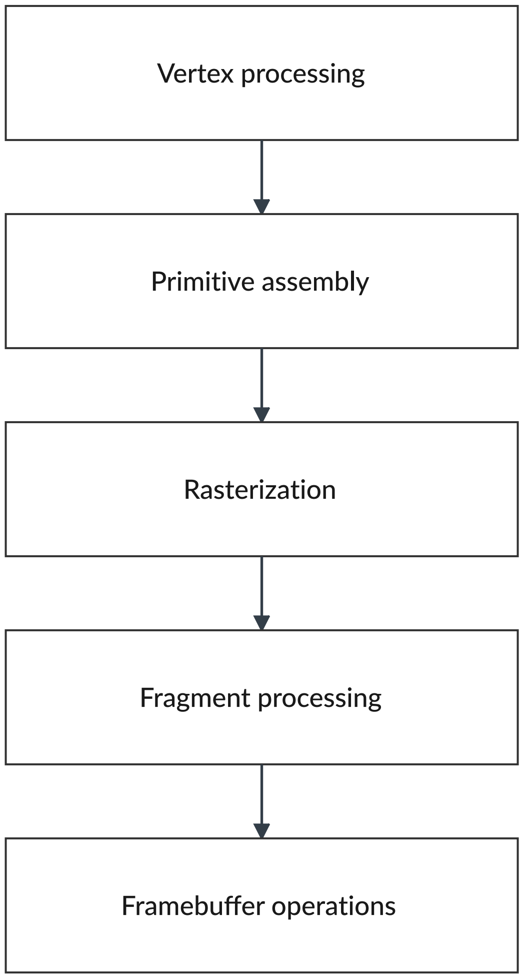
In the OpenGL ES 3.x pipeline, the shaders specify the vertex and fragment processing operations. The application must provide a pair of shaders for each draw call. A vertex shader defines the vertex processing operations and a fragment shader defines the fragment processing operations. The vertex shader is executed once per vertex, and the fragment shader is executed once per fragment.

Most of the semantics that are associated with data flowing through the pipeline are abstracted into the following special variables that are declared in the shaders:

- Generic vertex attributes. Generic vertex attributes replace all vertex data, such as position, normal vector, texture coordinates, and colors.
- Varying variables. All outputs from the vertex shader, except for position and point size, are abstracted into varying variables. These variables are interpolated across the primitive and are available to the fragment shader.
- Uniform variables. All global states that are required by vertex and fragment processing, such as transformation matrices, light positions, material properties, texture stage constants, and texture bindings, are abstracted into uniform variables. The application sets the values of these variables.

The following figure shows a simplified OpenGL ES graphics pipeline:

Figure 4-1: OpenGL ES graphics pipeline.



These stages are present, in different forms, in all versions of the OpenGL ES pipelines.

See the Mali Developer Center website for information on example shaders and sample code.

4.2 Primitive assembly

The Mali GPU generates primitives starting from the vertices.

A point contains one vertex, a line contains two vertices, a triangle contains three vertices. Vertices can be shared between multiple primitives, depending on the draw mode. If geometry or tessellation shaders are present, then vertices can generate a variable number of primitives.

4.3 Vertex processing

The vertex data provided by the application is read one vertex at a time, and the shader core runs a vertex shader program for each vertex.

This shader program performs:

- Lighting.
- Transforms.
- Viewport transformation.
- Perspective transformation.

The shader core or vertex processor also perform the following processing:

- Assembles vertices of graphics primitives.
- Builds polygon lists.

The output from vertex processing includes:

- The position of the vertex in the output framebuffer
- Additional data, such as the color of the vertex after lighting calculations.

4.4 Rasterization

Each primitive is divided into fragments so that there is one or more fragments for each pixel covered by the primitive.

4.4.1 Reads data

Reads the state information, polygon lists, and transformed vertex data. These are processed in a triangle setup unit to generate coefficients.

4.4.2 Rasterizes polygons

The rasterizer takes the coefficients from the triangle setup unit and applies equations to create fragments.

4.4.3 Interpolation

The properties at individual vertices are interpolated for each fragment produced by rasterization.

4.5 Fragment processing

Each fragment is assigned a color. This stage usually involves one or more texture look-ups. The fragment shader defines the fragment processing operations and it is executed once per fragment.

4.6 Framebuffer operations

The resulting pixel color is placed into the corresponding location in the framebuffer.

Depending on the render states set for the draw call, the fragment is subjected to a series of tests and combination operations on route to the location. The tests include:

- Depth testing. Compares the new fragment depth value to the depth value of the fragment that is previously written to this pixel, and discards it if it fails the depth test comparison.
- Blending. Calculates the resulting pixel color as a combination of the fragment color and the existing pixel color.
- Scissoring. Restricts rendering to a certain area of the framebuffer, and discards the fragment if it is located outside that area.

4.7 Framebuffer Fetch

The OpenGL ES API provides mechanisms for fetching framebuffer attachments on-tile. The Arm shader framebuffer fetch enables applications to read the current framebuffer color from the fragment shader, which is useful for techniques such as programmable blending.

Deferred shading-like techniques have evolved over time, but the fundamental remains where a G-buffer is rendered, and lighting is computed based on that G-buffer data.

This decouples geometry information from shading. Most of the recent innovation in the deferred space has revolved around reformulating the lighting pass. However, the fundamentals remain the same.

For example, the following images show a typical G-buffer layout of albedo, normals, material parameters like metallic-roughness, and depth.

Figure 4-2: G-buffer layout (Albedo).

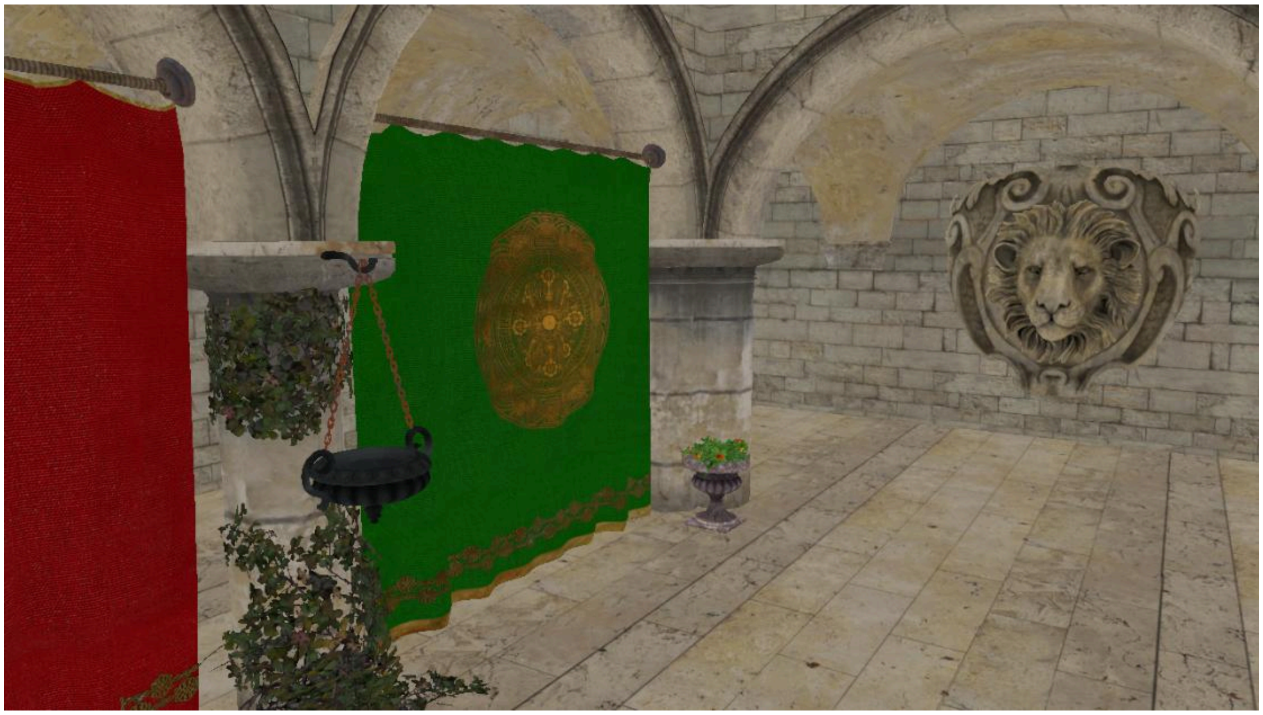


Figure 4-3: G-buffer layout (Normals).

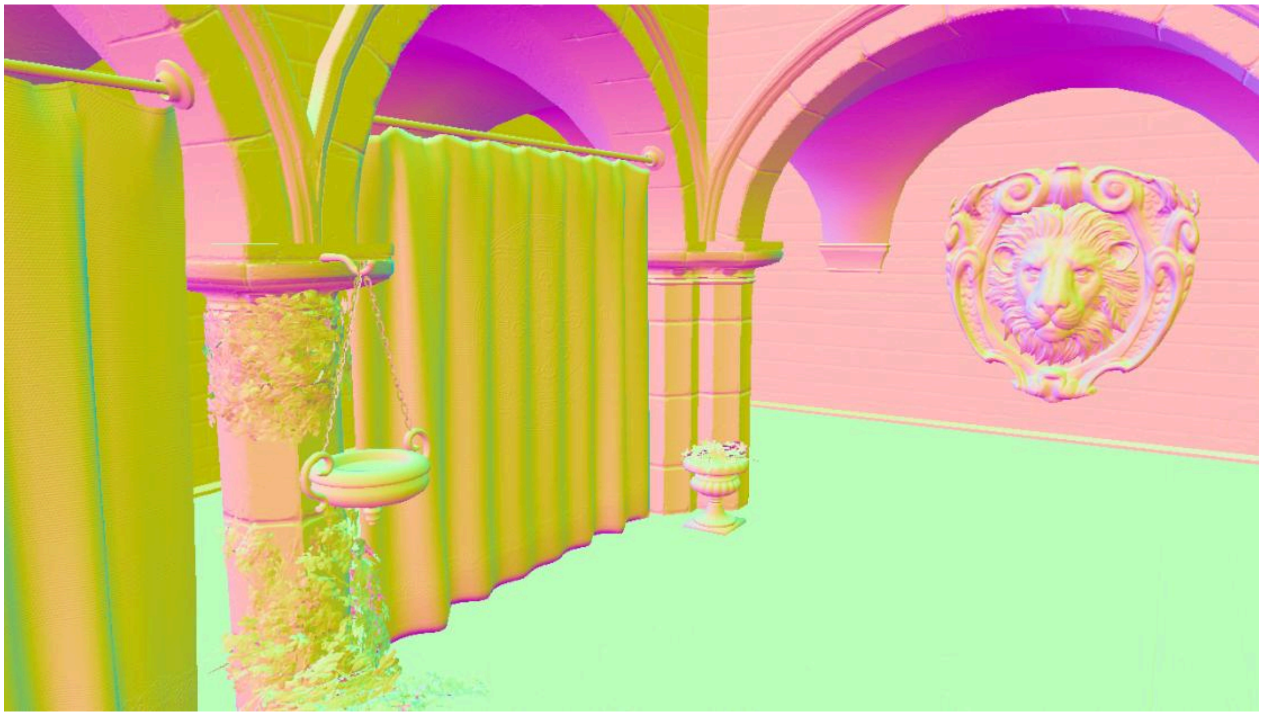


Figure 4-4: G-buffer layout (Material Parameters).

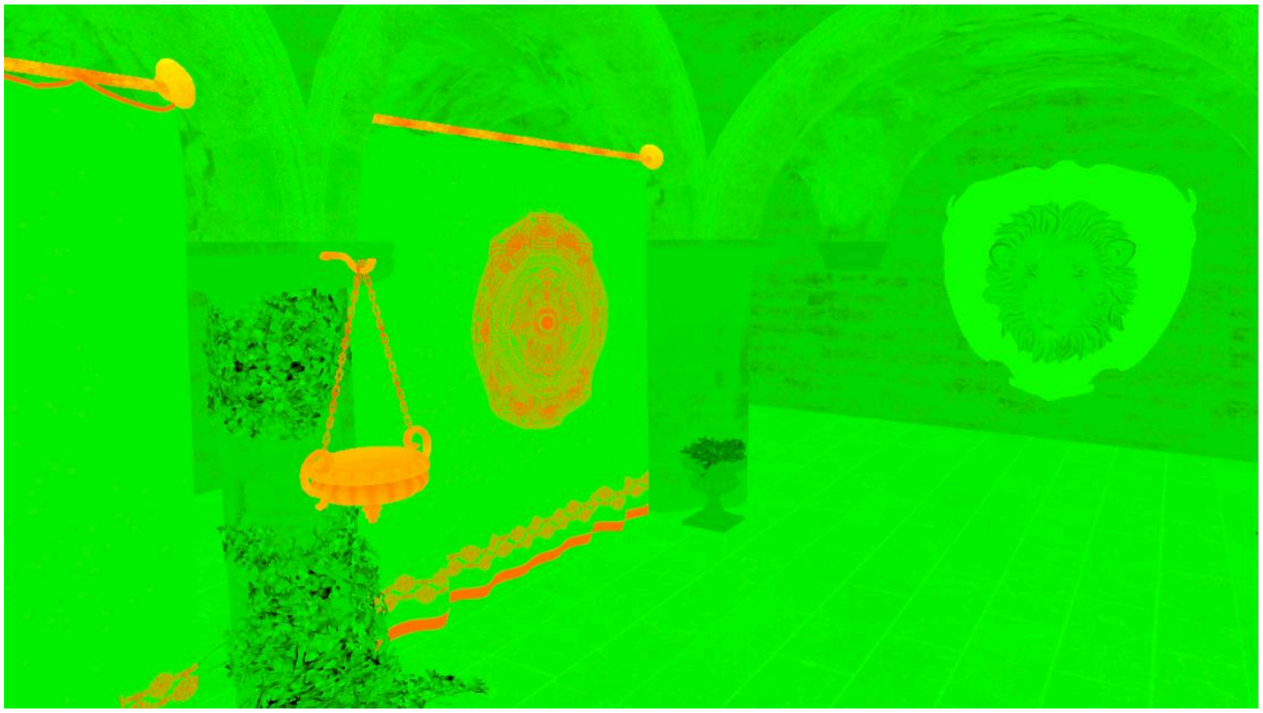
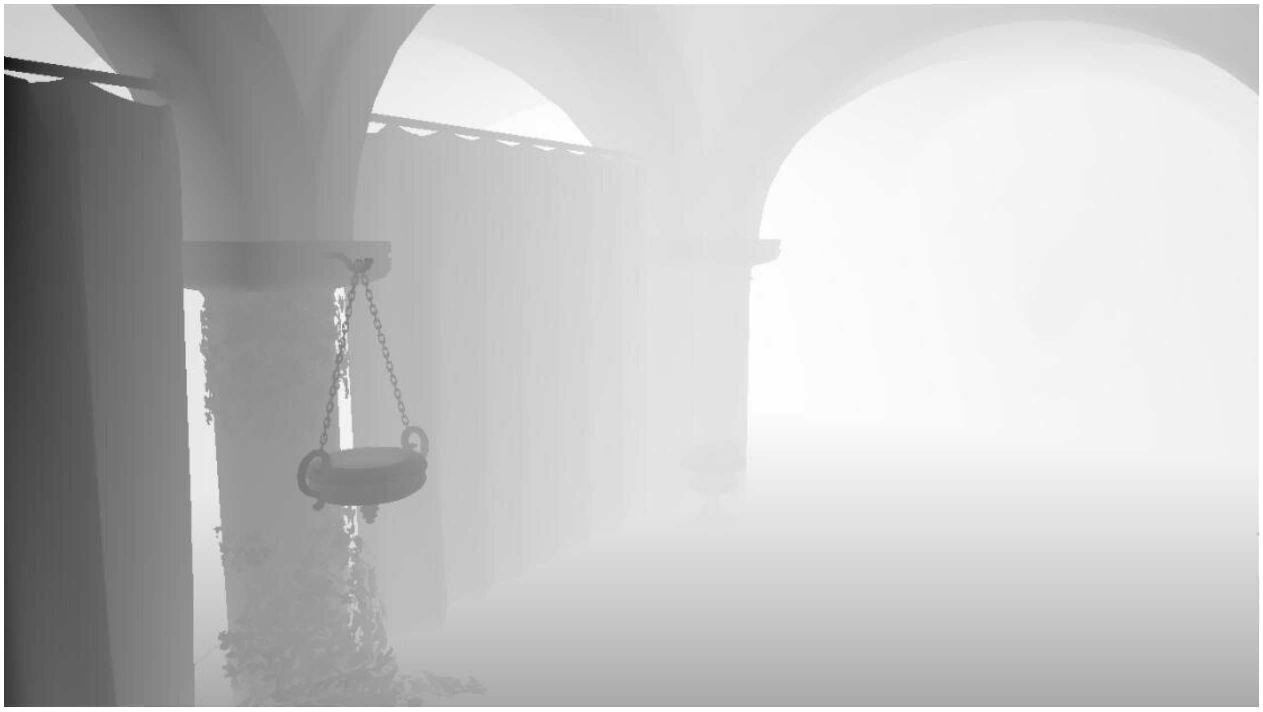


Figure 4-5: G-buffer layout (Depth).



These G-buffers are then used to shade the final pixel. In this example, rendering a trivial directional light.

4.7.1 The traditional implementation

The traditional way to implement this style of rendering is the use of *Multiple Render Targets* (MRT) with a G-buffer pass. This is then followed by a lighting pass, where the G-buffer is sampled with regular textures. This is still the conventional way on immediate mode GPUs, such as desktop GPUs.

Traditional G-buffer pass implementation

```
 #version 320 es
 precision mediump float;

 // B10G11R11_UFLOAT

layout(location = 0) out vec3 Emissive;

// RGBA8_SRGB

layout(location = 1) out vec3 Albedo;

// B10G10R10A2_UNORM

layout(location = 2) out vec3 Normal;

// R8G8_UNORM

layout(location = 3) out vec2 MetallicRoughness;


layout(location = 0) in highp vec2 vUV;

layout(location = 1) in vec3 vNormal;

layout(location = 2) in vec4 vTangent;


layout(binding = 0) uniform sampler2D TexAlbedo;

layout(binding = 1) uniform sampler2D TexNormal;

layout(binding = 2) uniform sampler2D TexMetallicRoughness;


void main()

{

    Albedo = texture(TexAlbedo, vUV).rgb;

    MetallicRoughness = texture(TexMetallicRoughness, vUV).xy;



    // Many different ways to implement this.

    vec2 tangent_xy = 2.0 * texture(TexNormal, vUV).xy - 1.0;

    float tangent_z = sqrt(max(0.0, 1.0 - dot(tangent_xy, tangent_xy)));

    vec3 tangent_normal = vec3(tangent_xy, tangent_z);

    vec3 bitangent = cross(vNormal, vTangent.xyz) * vTangent.w;

    mat3 TBN = mat3(vTangent, bitangent, vNormal);
    vec3 normal = normalize(TBN * tangent_normal);

    // [-1, 1] -> [0, 1] range.
    Normal = 0.5 * normal + 0.5;

    // This may or may not be relevant.

    // If present, we can reuse the lighting accumulation attachment.

    Emissive = vec3(0.0);

}
```

Traditional Lighting pass implementation

```
#version 320 es

precision mediump float;


// B10G11R11_UFLOAT

layout(location = 0) out vec3 Light;


layout(binding = 4) uniform Parameters

{

    highp mat4 inv_view_projection;
    highp vec2 inv_resolution;

};


// Textures we rendered to in G-buffer pass.

layout(binding = 0) uniform sampler2D GBufferAlbedo;
layout(binding = 1) uniform sampler2D GBufferNormal;
layout(binding = 2) uniform sampler2D GBufferMetallicRoughness;
layout(binding = 3) uniform sampler2D GBufferDepth;


vec3 compute_light(vec3 albedo, vec3 normal, vec2 metallic_roughness, highp vec3
 world)

{

    // Arbitrary complexity.
    ...

}

void main()

{

    ivec2 coord = ivec2(gl_FragCoord.xy);

    highp float depth = texelFetch(GBufferDepth, coord, 0).x;

    vec2 metallic_roughness = texelFetch(GBufferMetallicRoughness, coord, 0).xy;

    vec3 normal = 2.0 * texelFetch(GBufferNormal, coord, 0).xyz - 1.0;

    vec3 albedo = texelFetch(GBufferAlbedo, coord, 0).rgb;
    // Reconstruct world position.
    highp vec4 clip = vec4(2.0 * gl_FragCoord.xy * inv_resolution - 1.0,
                      depth, 1.0);
    highp vec4 world4 = inv_view_projection * clip;
    highp vec3 world = world4.xyz / world4.w;

    Light = compute_light(albedo, normal, metallic_roughness, world);
}
```

Blending is enabled for the color attachment, and you can render as many lights as you want, each with different shaders.

Early on, this kind of flexibility was a critical requirement for deferred shading, as shaders back in the mid-2000's had to be extremely simple and highly specialized.

However, the downside of this technique is the fill-rate, memory, and bandwidth requirements. Storing large G-buffers is costly, and shading multiple lights with overdraw is a big drain on bandwidth on immediate mode GPUs.

4.7.2 What tile-based architectures can optimize

With some help from the engine developer, there is a lot to gain by exploiting tile memory. For example, in the case of deferred shading, a straightforward implementation consumes a certain amount of bandwidth.

As a basic example, let us use normalized units and call them BU, where 1 BU represents `width x height x sizeof(pixel) bytes` of either reads or writes to external memory.

- Write G-buffer: 4 BU (albedo, normal, pbr, depth)
- Read G-buffer in lighting pass: >= 4 BU (could be more if caches are thrashed or lots of overdraw)
- Blend lighting buffer in lighting pass: >= 2 BU (1 BU read + 1 BU write, could be more with overdraw)

Cost: ~10 BU

4.7.3 Eliminating Bandwidth

The first consideration is that whenever `texelFetch` (`gl_FragCoord.xy`) appears, it could map directly to the current pixel's data.

Since you have access to framebuffer fetching mechanisms, you do not require the texture, so the shader code could be replaced with the hypothetical `readDataFromTile()`.

However, the assumption that pixels reside on-tile is only true if everything can happen in the same render pass. Therefore, to make tile-based GPUs work consecutively, the G-buffer and the lighting passes must happen after each other.

Writing a G-buffer attachment, or reading from it, occurs on-chip. Therefore, there is no external memory bandwidth costs.

Tile-based GPUs hold their framebuffers on-chip, and can choose whether the result actually needs to be flushed out to main memory.

This flush is the only significant bandwidth cost associated with framebuffer rendering and you can choose to skip it if required. Accessible by using `glInvalidateFramebuffer()`.

4.7.4 Theoretical optimal saving

In the best case scenario, you only have 1 BU of bandwidth available. Therefore, write the final shaded light attachment, which returns around a 10x reduction in bandwidth usage.

4.7.5 Do these assumptions work?

This type of optimization is based on some key assumptions. There are various scenarios which can break them, and you then have to settle for a less efficient optimization as a result. As a developer, you must weigh these losses against your desired image quality.

5 Pixel Local Storage

This chapter describes *Pixel Local Storage* (PLS), how to use PLS, and provides some examples of how to implement PLS.

5.1 About Pixel Local Storage

In 2014, Arm introduced the *Pixel Local Storage* (PLS) OpenGL ES extension. Since then, many things have happened that have transformed mobile graphics, particularly the release of the first version of Vulkan in February 2016.

Built from the ground up, Vulkan was intended to replace OpenGL as the main graphics API. OpenGL had successfully served the industry for more than 20 years, but it was time for a clean start. The new graphics API was expected to provide a set of benefits across multiple platforms that the graphics community recognizes and values today.

While the default API in the main game engines is now Vulkan, every engine still supports OpenGL ES 3.x for Android. OpenGL ES 3.1 brought compute to mobile graphics, and OpenGL ES 3.2 added the Android Extension Pack, bringing the mobile API's functionality significantly closer to its desktop version – OpenGL.

OpenGL ES 3.2 is supported in Android 6.0 and higher - if the device itself supports this graphics pipeline. Almost 75% of Android devices support OpenGL ES 3.2, which is reflected in the high use of OpenGL ES by game developers.

5.1.1 Advantages of PLS

PLS saves memory bandwidth usage and increases performance.

Mali GPUs render images in either 16x16 or 32x32 pixel tiles. During processing, the data for this is stored in a tile buffer in the GPU.

The PLS extension enables your shaders to directly access the tile buffer, so they can store data. By storing data on the GPU, your shaders can store data and pass data between shaders without accessing external memory.

Memory bandwidth consumes a lot of power in mobile devices so saving memory bandwidth is very useful. Mobile systems have bandwidth and power limitations so using too much bandwidth reduces performance.

With PLS, you can chain different graphics techniques together without accessing external memory. This lets you use techniques from desktop platforms such as deferred rendering that would otherwise not be easy to implement effectively on mobile devices.

5.1.2 Operation of PLS

Depending on the model of Mali GPU hardware being used, PLS provides either 128 or 256 bits of storage per pixel that is shared by all the fragments that cover the pixel. You can utilize this storage any way you want to, and you are not limited to the OpenGL ES formats. For example, you can store color, normals, depth, and other information.

PLS is stored in the same buffer as color data so writing one overwrites the other. The depth and stencil data are stored separately so these are not affected.

When the Mali GPU has processed a tile, it discards the PLS data. This means it only writes the tile out to memory and does not use any additional memory bandwidth. Each shader uses the PLS memory in turn and declares its own view of the PLS memory. This means that each shader can interpret the memory in whatever way suits it best. The PLS declaration is independent of the framebuffer format.

To generate output, each of your shaders builds up information in turn in the PLS. You create a final custom shader known as a resolve shader, to generate the color information, and then write it to the framebuffer in the correct format.

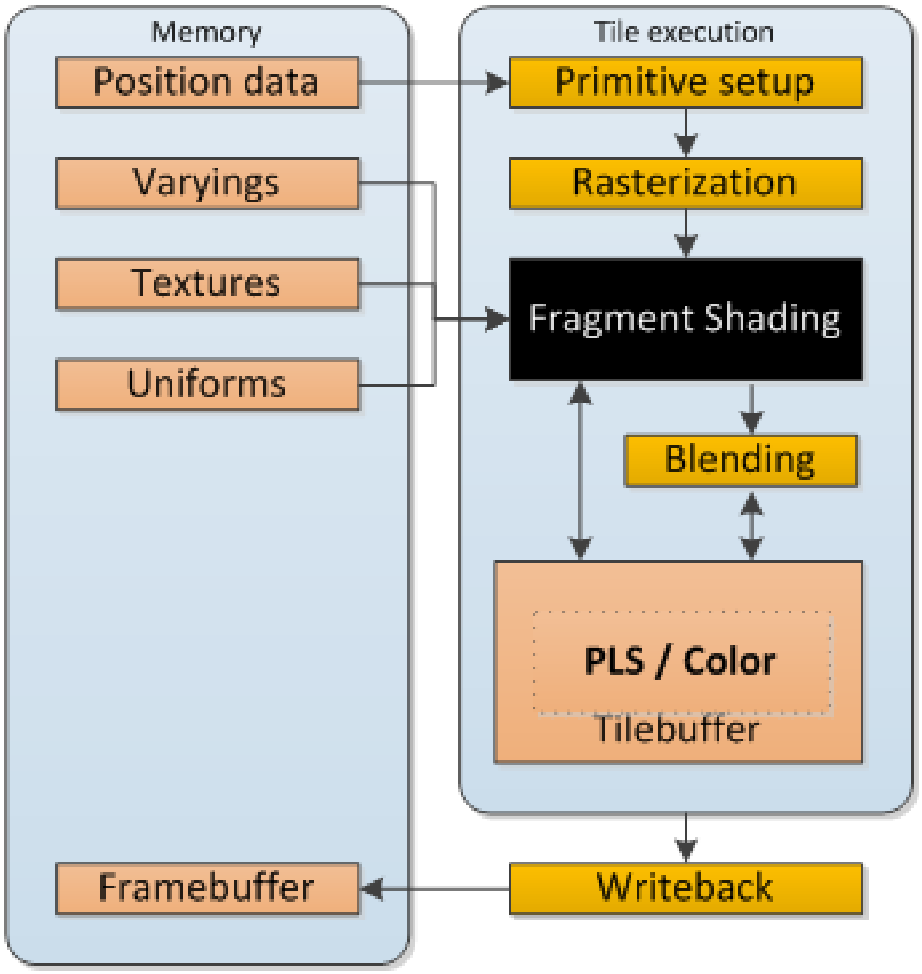
5.2 Using Pixel Local Storage

PLS takes advantage of the tile-based rendering approach used by the Mali GPU. Mali GPUs break up the screen into small regions of 16x16, or 32x32, pixels known as tiles. Rendering takes place in two passes. The first pass builds the list of geometric primitives that fall into each tile. Then in the second pass, each shader core executes the fragment shading tile-by-tile and then writes these tiles back to memory as they have been completed.

This approach keeps the size of each tile small. During shading it is possible to keep the color, depth, and stencil data in an on-chip RAM within the shader core. This RAM is fast and is tightly coupled to the GPU shader core, allowing you to save valuable bandwidth and therefore power.

The following diagram shows how the pipeline is modified by PLS:

Figure 5-1: OpenGL ES pipeline modified by PLS.



Originally, per-pixel on-chip memory storage on a Mali GPU was normally used to do multisampling anti-aliasing. PLS is an API extension that exposes this memory to the programmer so they may read or even write their own per-pixel data.

This data is preserved between draw calls and remains active as long as the framebuffer remains active. This persistent per-pixel storage was the key property that made PLS unique, changing the way graphics can be achieved on tile-based GPUs.

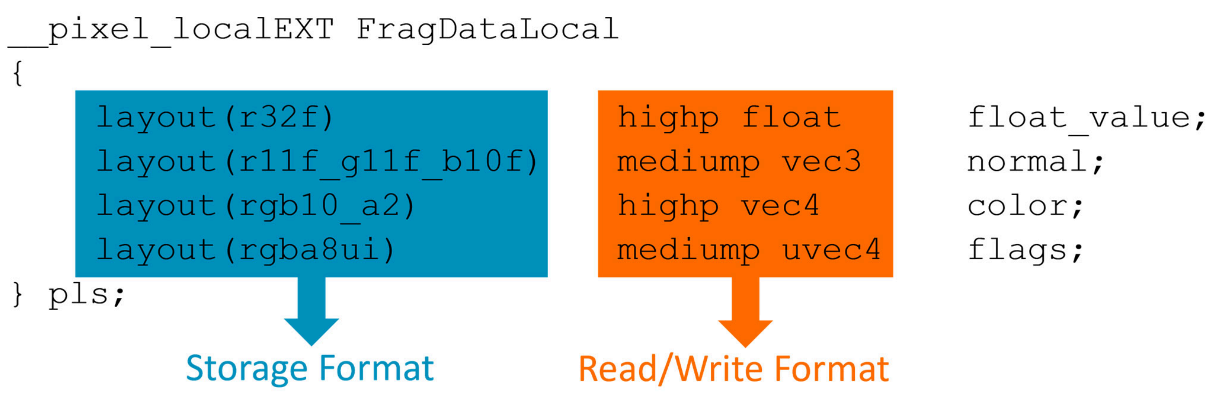
With PLS it became possible to chain render tasks without flushing out the memory, keeping bandwidth consumption down massively. You would typically use this memory to build up the final pixel color progressively, using multiple shaders, with a final resolve shader at the end to explicitly copy to the framebuffer output.

PLS allows each shader to declare its per-pixel view of the PLS as a struct. This allows you to re-interpret the data and change the view between shaders.

The per-pixel view of the PLS is independent of the current framebuffer format, meaning that what is flushed back to main memory in the end still conforms to the current framebuffer format.

The following image shows an example of a PLS shader view:

Figure 5-2: PLS structure.



```
__pixel_localEXT FragDataLocal
{
    layout(r32f)              highp float     float_value;
    layout(r11f_g11f_b10f)    mediump vec3    normal;
    layout(rgb10_a2)          highp vec4      color;
    layout(rgba8ui)           mediump uvec4   flags;
} pls;
```

The layout qualifier on the left is used to specify the data format of the individual PLS variables, with all formats 32-bits in size. The precision and type specified in the middle describes the type that the shader uses to read and write to these variables.

There is an implied conversion between this type and the layout format when you read and write from your shader. For more detailed information we recommend reading the extension specification in the Additional Reading section.

Before PLS, GLES 3.0 relied on Multiple Render Targets (MRT) for complex rendering tasks, which allowed the shader output to be written to more than one texture in a single render pass. These textures can then be used as inputs to other shaders. A common use of MRT in OpenGL is deferred rendering that performs lighting calculations of the entire 3D scene at once, compared to each individual object as happens in forward rendering.

Deferred rendering is based on the idea that most of the lighting is deferred or postponed to a later stage. This approach significantly optimizes scenes with large numbers of lights and is commonly used in consoles. In the first render pass, known as the geometry pass, the scene is rendered once to retrieve geometrical information from the objects, which is stored in a collection of textures called the G-buffer.

MRT is used to store the information for the lighting calculations in multiple render targets, which are then used after the entire scene has been drawn to calculate the final lit image. Typically, one render target stores color and surface information of objects, while another holds the surface normals and depth information. Additional render targets can be used to store ambient occlusion data.

The problem with using MRT on mobile devices is that each render target is written to the main memory and is then read back in the next step to retrieve the stored information.

Although the combination of MRT with framebuffer fetch looks like a useful alternative, PLS has some extra benefits, including:

- Many fewer performance pitfalls,
- More flexible storage as the data format is independent of the color attachment format,
- Offers a programming model that is closer to what the shader programmer wants to express.

5.3 Examples of Pixel Local Storage

Pixel Local Storage enables the use of techniques that might not otherwise be possible on mobile devices.

5.3.1 Deferred shading

Deferred rendering can be implemented efficiently on mobile using the on-chip memory exposed by the PLS extension. Initially, the available tile memory was 128 bits/pixel.

This is the case for the Mali GPUs released under Midgard 3rd and 4th micro-architecture generations and the Arm Mali-G71 GPU released under the next-generation Bifrost architecture.

Since then, the tile memory available for PLS has been doubled to 256 bits/pixels:

Mali GPU Models with 128 Bits/pixel:

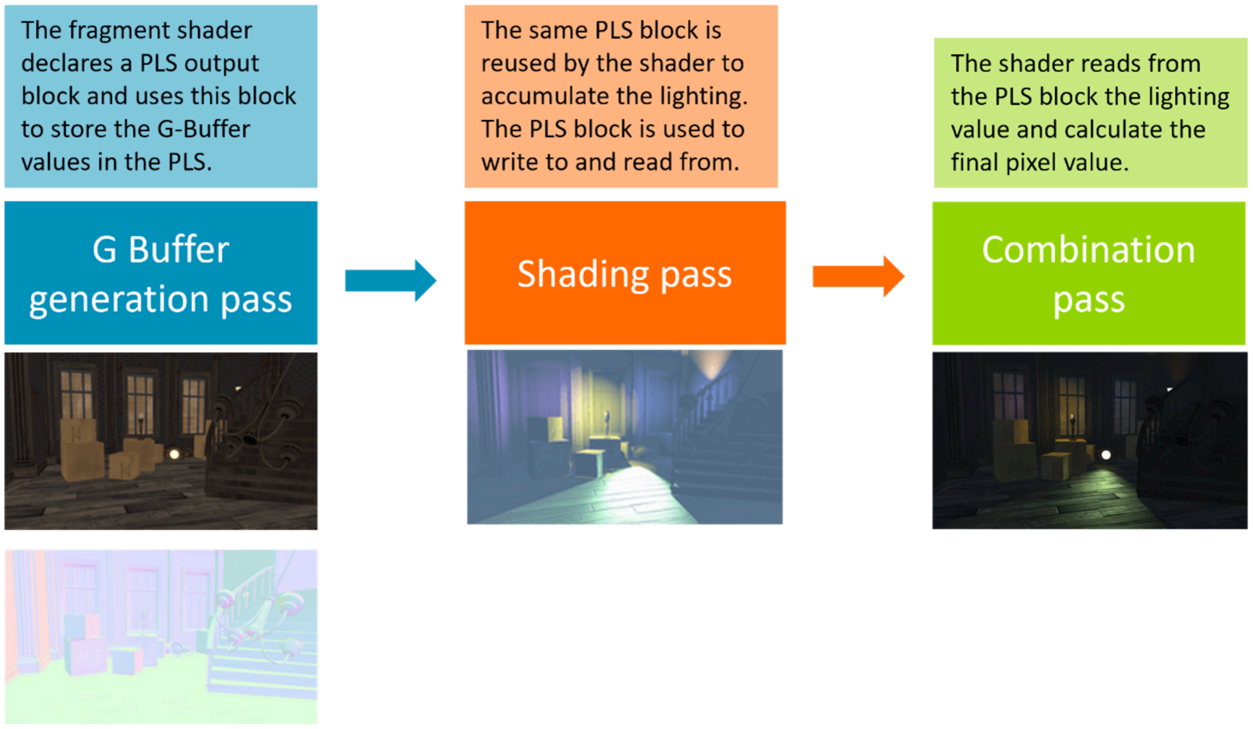
- T720
- T760
- T820
- T830
- T860
- T880
- G71

Mali GPU Models with 256 Bits/pixel:

- G72
- G31
- G51
- G52
- G76
- G57
- G77
- G68
- G78

The following image summarizes how deferred rendering can be implemented using PLS in three passes:

Figure 5-3: Deferred rendering passes using PLS.


The fragment shader declares a PLS output block and uses this block to store the G-Buffer values in the PLS.
G Buffer generation pass
The same PLS block is reused by the shader to accumulate the lighting. The PLS block is used to write to and read from.
Shading pass
The shader reads from the PLS block the lighting value and calculate the final pixel value.
Combination pass

Note

With Vulkan's multipass concept, it is not possible to directly access tile storage. Instead, we provide enough information upfront to the driver, so that it can optimize a render pass to allow the current pass to access the pixel data stored by the previous pass at the same location. The same approach that PLS uses.

In the G-Buffer pass, the fragment shader declares a PLS output block, which is then filled by the shader with both the scene color and normals information. Note that the total memory used by the PLS output block is 128 bits (32 bits x 4).

```
__pixel_local_outEXT FragData
{
    layout(rgba8) highp vec4 Color;
    layout(rg16f) highp vec2 NormalXY;
    layout(rg16f) highp vec2 NormalZ_LightingB;
    layout(rg16f) highp vec2 LightingRG;
} gbuf;

void main()
{
    gbuf.Color = calcDiffuseColor();
    vec3 normal = calcNormal();
    gbuf.NormalXY = normal.xy;
    gbuf.NormalZ_LightingB.x = normal.z;
}
```

The extension qualifier `__pixel_local_outEXT` means that the storage can be written to and is persistent across shader invocations that cover the same pixel. The storage is then read in another shader invocation that declares either `__pixel_localEXT` or `__pixel_local_inEXT` storage.

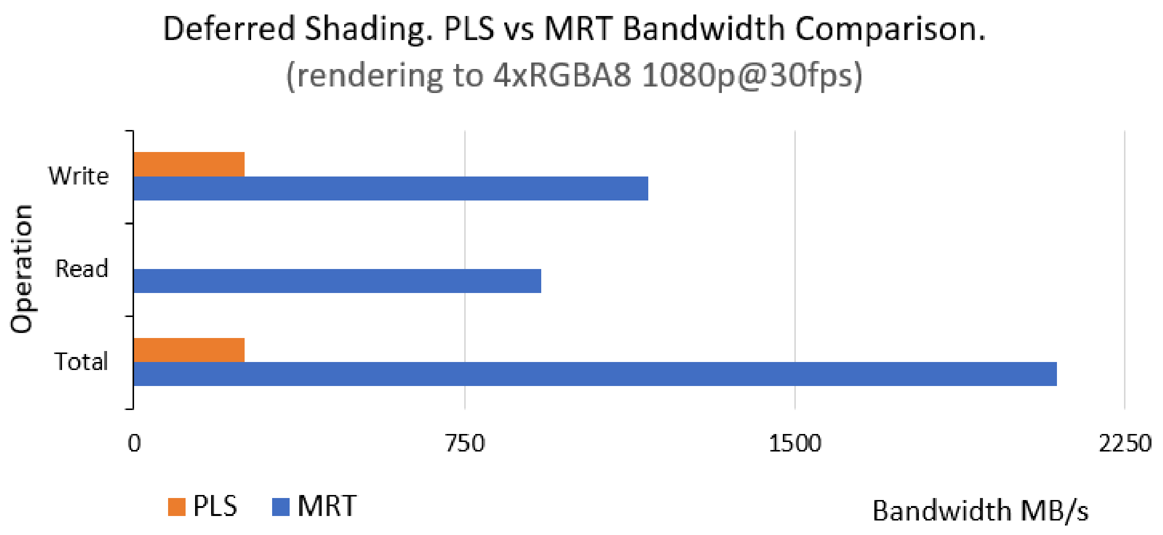
In the second pass, the same PLS is declared. However, this pass uses the qualifier `__pixel_localEXT` to indicate that the storage can be both read and written. The shader reads the stored color and normal data from the previous pass and then writes the calculated lighting. At this point, the PLS block contains the color, normal, and accumulated lighting information.

```
void main()
{
    vec3 lighting = calclighting(gbuf.NormalXY.x,
                                 gbuf.NormalXY.y,
                                 gbuf.NormalZ_LightingB.x);
    gbuf.LightingRG += lighting.xy;
    gbuf.NormalZ_LightingB.y += lighting.z;
}
```

The third pass only reads the lighting information and calculates the final pixel color value from the PLS block. It is worth noting that the PLS is automatically discarded once the tile is fully processed, so it has no impact on external memory bandwidth. The only data that goes off-chip is the data explicitly copied to the out variables, at which point the PLS data is invalidated.

The following image shows a bandwidth consumption comparison between PLS and MRT:

Figure 5-4: PLS vs MRT bandwidth consumption in deferred shading.



The PLS approach consumes ~8x less bandwidth than the MRT approach - which is a significant difference. Less bandwidth consumption means less power and therefore results in a longer battery life. It is worth noting that the impact of bandwidth reduction on power saving is probably even more significant if we consider improved thermals as part of the equation.

Less bandwidth consumption also means higher performance as less time is spent on data traffic. A benchmark study which was devoted to on-chip memory management, shows that keeping data on-chip can result in a 40% performance improvement.

Benchmark Study: Investigation: Managing on-chip memory (GitHub)

5.4 Advanced Shading Techniques with Pixel Local Storage

This section shows some examples of the output of different techniques that use *Pixel Local Storage* (PLS).

There are several advanced techniques that you can use when utilizing PLS. They include:

5.4.1 Translucency

Before PLS, translucency-like effects required several passes with the consequent memory flush out and data read back from main memory. Bandwidth consumption was a practical barrier for many complex rendering techniques on mobile.

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 a common example of translucent materials. To realistically render these materials, we need to consider *Sub Surface Scattering* (SSS).

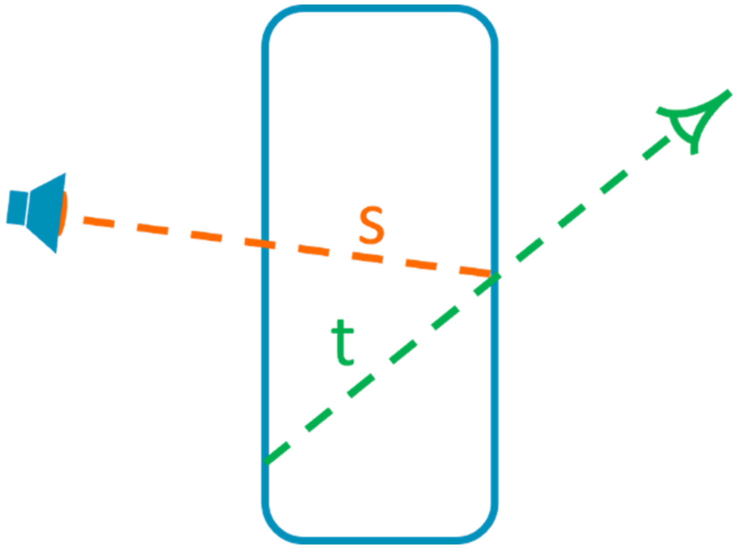
This is the light transport mechanism that explains how the light that penetrates the surface of a translucent object is scattered by interacting with the material and exiting the surface at a different point. We all easily recognize this effect, for example, when light passes through ears.

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, like 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).

The following image shows the approximate light attenuation considering thickness as viewed from the camera:

Figure 5-5: Light attenuation.



Following this approach, compute the thickness as the difference between the maximum and the minimum view-space depth of an object 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 following PLS block is then allocated. Please note that the total size of the structure is 128 bits:

```
    __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:

```
    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;
    }
```

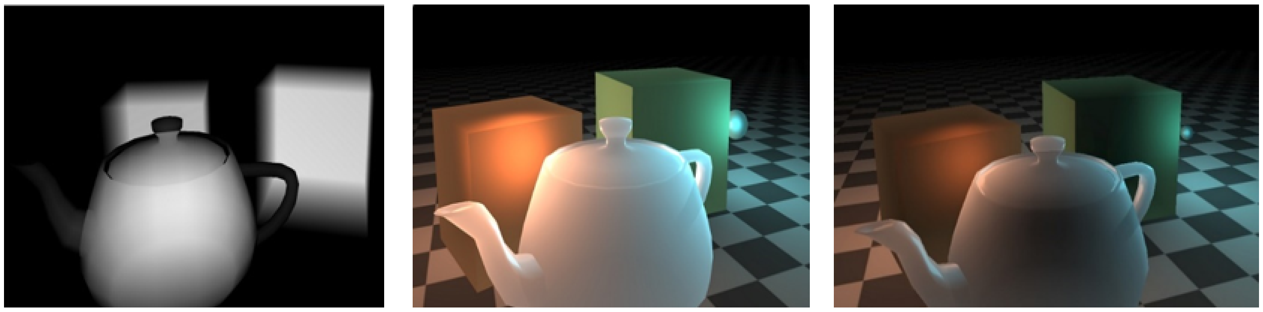
In the second pass, the same PLS block is used, and the shader finds the maximum depth of the previously determined closest object. This time the scene is rendered without depth testing, but with a stencil test set to equal each object's ID.

Since the ID of the closest object is stored in the stencil buffer, only that same object will have fragments that pass through:

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

The following image on the shows the rendering of the resulting thickness of the translucent objects (left), the translucency rendering (center) and the lower-intensity light rendering (right):

Figure 5-6: 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.

In the translucency example, the middle of Figure 5-6, two lights move in the opposite direction, both going through the cubes. By changing some parameters in the translucency example we can see the resulting altered effect, as noted in the picture on the right.

The following sample uses OpenGL ES 3.0 and PLS to perform advanced shading techniques. The sample computes a per-pixel object thickness, and uses it to render a subsurface scattering effect for translucent geometry, without the use of external depth-maps or additional render targets.

Advanced Shading Techniques with PLS

5.4.2 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, our 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 amount and the complexity of the semi-transparent geometry used, the ordering can take a significant amount of time and not always produce the right result. Alternative approaches known as *Order Independent Transparency* (OIT) 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 in bounded memory and is 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 then merge when required. Ensuring that the strict ordering requirements of alpha blending can be relaxed.

For the simplest case of two layers, the blended layers can be built-up 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.

SIGGRAPH 2014 presentation (PDF): Efficient Rendering with Tile Local Storage Presentation.

6 Related extensions

Extensions enable your shaders to read from the framebuffer, framebuffer depth, and stencil values.

6.1 Extension: GL_ARM_shader_framebuffer_fetch

This extension enables your shader to read the existing framebuffer value.

To enable this extension, you must add the following code to the beginning of your fragment shader.

```
#extension GL_ARM_shader_framebuffer_fetch : enable
```

6.1.1 Variable for GL_ARM_shader_framebuffer_fetch extension

Variable name: `gl_LastFragColorARM`

Type: vec4

Description: Reads the existing framebuffer color.

6.1.2 Extension: GL_ARM_shader_framebuffer_fetch_depth_stencil

This extension enables your shader to read the existing framebuffer depth, and stencil values.

To enable this extension, you must add the following code to the beginning of your fragment shader.

```
#extension GL_ARM_shader_framebuffer_fetch_depth_stencil : enable
```

6.1.3 Variables for GL_ARM_shader_framebuffer_fetch_depth_stencil extensions

Variable name: `gl_LastFragDepthARM`

Type: float

Description: Reads the existing framebuffer depth value. The value required is in the window coordinate space.

Variable name: `gl_LastFragStencilARM`

Type: int

Description: Reads the existing framebuffer stencil value.

7 Adaptive Scalable Texture Compression ASTC texture compression

You can find the latest version of the developer guide for Arm's *Adaptive Scalable Texture Compression* (ASTC) on the Arm Developer website:

Learn the basics, ASTC developer guide.

8 Compute shaders

This chapter details compute shaders, what work groups are, and provides a compute shader example.

8.1 About compute shaders

Compute shaders enable you to implement complex algorithms and make use of GPU parallel programming.

Compute shaders enable you to use GPU compute in the same OpenGL ES API and shading language you use for graphics rendering.

You are not required to learn another API to use GPU compute. The compute shader is another type of shader in addition to the existing vertex and fragment shaders.

8.1.1 Differences between compute shaders, vertex and fragment shaders

There are several differences between compute shaders, vertex and fragment shaders.

8.1.2 Compute shaders features

Compute shaders are general purpose and are less restricted in their operation compared to vertex and fragment shaders.

Compute shaders include the following features:

- Compute shader threads correspond to iterations of a nested loop, rather than to graphics constructs like pixels or vertices.
- Compute shaders are not part of the graphics pipeline, they have a dedicated single stage pipeline that executes independently of the rest of the pipeline. You are not required to think about the pipeline stages, as you are with vertex and fragment shaders.
- Compute shaders are based on the existing *OpenGL Shading Language* (GLSL) syntax. This makes it easy for OpenGL ES developers to learn how to write compute shaders.
- Compute shaders are not restricted by the inputs and outputs of specific pipeline stages. They have direct random access to memory and can read and write arbitrary data that is stored in memory buffers or texture images.
- Compute shaders can write to *Shader Storage Buffer Objects* (SSBO). SSBOs provide a flexible input and output option and enable you to exchange data between pipeline stages. A typical use case for compute shaders it to create data that is used by the graphics pipeline.
- Explicit parallelism provides you with full control over how many threads are created.

You can consider compute shaders as a lightweight alternative to OpenCL. They are easier to use and integrate with the rest of OpenGL ES.

Vertex and fragment shaders features

Vertex and fragment shaders can perform computations but this ability is limited because they are designed for specific purposes.

The graphics pipeline is flexible but it places some restrictions on vertex and fragment shaders:

- Vertex and fragment shaders can read data from arbitrary locations, but can only write to specific memory locations and data types:
    - A fragment shader can only write to one specific pixel in a texture.
    - A vertex shader can only write to vertex slot in the transform feedback.
- Vertex and fragment shaders can only read or write in specific pipeline stages. You must consider this when developing them.
- Parallelism is implied:
    - The GPU and driver controls the parallelism.
    - The driver parallelizes the workload, you have no control over how this happens.
- Threads are independent:
    - There is no sharing of data.
    - There is no synchronization between threads.

8.2 Work groups

Compute shader threads are arranged into work groups.

A single work group consists of multiple threads that you define in the shader code.

Work groups must support at least 128 threads, but GPUs can optionally support more. You can query this with the appropriate `glGetIntegerv()` function calls.

You can arrange work groups in one to three dimensions. The different dimensions are useful because you can use them for different types of computations, for example:

- 1D for audio processing.
- 2D for image processing.
- 3D for volumetric processing.

Work groups are independent of each other.

You can synchronize and share data between threads if they are in the same work group. This enables you to avoid using expensive multi-pass techniques. You cannot synchronize execution between threads in different work groups.

Compute shaders include a set of synchronization primitives. These enable you to control the ordering of execution and memory accesses by the different threads running in parallel on the GPU. This ensures that results do not depend on the order that the threads run in.

8.3 Compute shaders example

A simple example of how to implement compute shaders within your application.

This example calculates a colored circle with a given radius. The radius is a uniform parameter passed by the application and is updated every frame to animate the radius of the circle.

The whole circle is drawn using points that are stored as vertices within a *Vertex Buffer Object* (VBO). The VBO is mapped onto an SSBO without any extra copying in memory, and passed to the compute shader.

The following code is the compute shader code:

```
        #version 310 es

        // The uniform parameters that are passed from application for every frame.
        uniform float radius;

        // Declare the custom data type that represents one point of a circle.
        // This is vertex position and color respectively,
        // that defines the interleaved data within a
        // buffer that is Vertex|Color|Vertex|Color|
        struct AttribData
        {
            vec4 v;
            vec4 c;
        };

        // Declare an input/output buffer that stores data.
        // This shader only writes data into the buffer.
        // std430 is a standard packing layout which is preferred for SSBOs.
        // Its binary layout is well defined.
        // Bind the buffer to index 0. You must set the buffer binding
        // in the range [0..3]. This is the minimum range approved by Khronos.
        // Some platforms might support more indices.
        layout(std430, binding = 0) buffer destBuffer
        {
            AttribData data[];
        } outBuffer;

        // Declare the group size.
        // This is a one-dimensional problem, so prefer a one-dimensional group
layout.
        layout (local_size_x = 64, local_size_y = 1, local_size_z = 1) in;

        // Declare the main program function that is executed once
        // glDispatchCompute is called from the application.
        void main()
        {
            // Read the current global position for this thread
            uint storePos = gl_GlobalInvocationID.x;

            // Calculate the global number of threads (size) for this work dispatch.
            uint gSize = gl_WorkGroupSize.x * gl_NumWorkGroups.x;

            // Calculate an angle for the current thread
            float alpha = 2.0 * 3.14159265359 * (float(storePos) / float(gSize));

            // Calculate the vertex position based on
            // the previously calculated angle and radius.
            // This is provided by the application.
            outBuffer.data[storePos].v = vec4(sin(alpha) * radius, cos(alpha) *
radius, 0.0, 1.0);

            // Assign a color for the vertex
            outBuffer.data[storePos].c = vec4(float(storePos) / float(gSize), 0.0,
1.0, 1.0);
        }
```

When you have written the compute shader code, you must make it work in your application. Within the application, you must create the compute shader. This is just a new type of shader `GL_COMPUTE_SHADER`. The other calls related to the initialization remain the same as for vertex and fragment shaders.

The following code creates the compute shader and checks for compilation and linking errors:

```
        // Create the compute program the compute shader is assigned to
        gComputeProgram = glCreateProgram();

        // Create and compile the compute shader.
        GLuint mComputeShader = glCreateShader(GL_COMPUTE_SHADER);
        glShaderSource(mComputeShader, 1, computeShaderSrcCode, NULL);
        glCompileShader(mComputeShader);

        // Check if there were any issues compiling the shader.
        int rvalue;
        glGetShaderiv(mComputeShader, GL_COMPILE_STATUS, &rvalue);

        if (!rvalue)
        {
            glGetShaderInfoLog(mComputeShader, LOG_MAX, &length, log);
            printf("Error: Compiler log:\n%s\n", log);
            return false;
        }

        // Attach and link the shader against the compute program.
        glAttachShader(gComputeProgram, mComputeShader);
        glLinkProgram(gComputeProgram);

        // Check if there were any issues linking the shader.
        glGetProgramiv(gComputeProgram, GL_LINK_STATUS, &rvalue);

        if (!rvalue)
        {
            glGetProgramInfoLog(gComputeProgram, LOG_MAX, &length, log);
            printf("Error: Linker log:\n%s\n", log);
            return false;
        }
```

When you have created the compute shader on the GPU, you must set up handlers that are used for setting up inputs and outputs for the shader.

In this case, you must retrieve the radius uniform handle and set the integer variable `gIndexBufferBinding` to `0` because the binding is hard-coded with `binding = 0`.

Using this index, you can bind the VBO to the index and write data from within the compute shader to the VBO:

```
        // Bind the compute program so it can read the radius uniform location.
        glUseProgram(gComputeProgram);

        // Retrieve the radius uniform location
        iLocRadius = glGetUniformLocation(gComputeProgram, "radius");

        // See the compute shader: "layout(std140, binding = 0) buffer destBuffer"
        gIndexBufferBinding = 0;

        Start the compute shader and write data to the VBO. The following code shows
how to bind the VBO to the SSBO and submit a compute job to the GPU:

        // Bind the compute program.
        glUseProgram(gComputeProgram);

        // Set the radius uniform.
        glUniform1f(iLocRadius, (float)frameNum);

        // Bind the VBO to the SSBO, that is filled in the compute shader.
        // gIndexBufferBinding is equal to 0. This is the same as the compute shader
binding.
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, gIndexBufferBinding, gVBO);

        // Submit a job for the compute shader execution.
        // GROUP_SIZE = 64
        // NUM_VERTS = 256
        // As the result the function is called with the following parameters:
        // glDispatchCompute(4, 1, 1)
        glDispatchCompute(NUM_VERTS / GROUP_SIZE_WIDTH, 1, 1);

        // Unbind the SSBO buffer.
        // gIndexBufferBinding is equal to 0. This is the same as the compute shader
binding.
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, gIndexBufferBinding, 0);
```

ou pass the number of groups to be executed to the `glDispatchCompute()` function, not the number of threads. In this example this is `2 x 2 x 1` groups, giving 4. However, the real number of threads executed is `4 x [8 x 8]` and this results with a total number of 256 threads. The `8 x 8` numbers are from the compute shader source code where they are hard-coded.

The compute shader jobs are dispatched and the compute shaders update the VBO buffer. When this is complete, you render the results to the screen.

All jobs are submitted and executed on the Mali GPU in parallel. You must ensure the compute shaders jobs are finished before the draw command starts fetching data from the VBO buffer.

Note

This is a special property of compute shaders. You do not usually have to consider this in OpenGL ES.

```
        // Call this function before you submit a draw call,
        // that uses a dependency buffer, to the GPU
        glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT);

        // Bind the VBO
        glBindBuffer( GL_ARRAY_BUFFER, gVBO );

        // Bind the vertex and fragment rendering shaders
        glUseProgram(gProgram);
        glEnableVertexAttribArray(iLocPosition);
        glEnableVertexAttribArray(iLocFillColor);

        // Draw points from the VBO
        glDrawArrays(GL_POINTS, 0, NUM_VERTS);
```

To present the VBO results on screen, you can use the following vertex and fragment programs.

The following code shows the vertex shader:

```
        attribute vec4 a_v4Position;
        attribute vec4 a_v4FillColor;
        varying vec4 v_v4FillColor;

        void main()
        {
            v_v4FillColor = a_v4FillColor;
            gl_Position = a_v4Position;
        }
```

The following code shows the fragment shader:

```
        varying vec4 v_v4FillColor;

        void main()
        {
            gl_FragColor = v_v4FillColor;
        }
```

9 Geometry shaders

This chapter details what geometry shaders are, isosurfaces, the implementation of surface nets on a Mali GPU, and we provide an example use for gemoetry shaders.

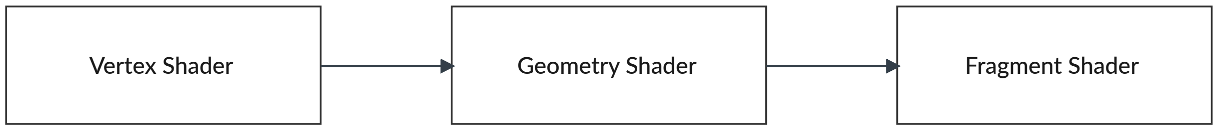
9.1 About geometry shaders

Geometry shaders add an optional programmable stage in the graphics pipeline that enables the creation of new geometry from the output of the vertex shader.

The ability to create geometry shaders is included in OpenGL ES 3.2 and the *Android Extension Pack* (AEP) for OpenGL ES 3.1.

The following figure shows the position of geometry shaders in the graphics pipeline.

Figure 9-1: Geometry shader pipeline position.



Geometry shaders use a function that operates on input primitives. Geometry shaders produce geometry in the shape that the function describes. Altering the function produces different results.

Expanding geometry using a geometry shader can reduce bandwidth requirements because the bandwidth required for memory reads is reduced compared to traditional methods.

9.1.1 A basic geometry shader example

A geometry shader example takes N points and produces an output from them.

The following code shows an example draw call that performs this action:

```
ayout(points) in;
in vec4 vs_position[];
void main()
{
    vec4 position = vs_position[0];
    gl_Position = position - vec4(0.05, 0.0, 0.0);
    EmitVertex();
    gl_Position = position + vec4(0.05, 0.0, 0.0);
    EmitVertex();
    gl_Position = position + vec4(0.0, 0.05, 0.0);
    EmitVertex();
    EndPrimitive();
}
```

The result from a geometry shader is stored in texture memory, this makes it very compact.

9.1.2 Related information

OpenGL Geometry page.

9.2 Isosurfaces

A geometry shader creates an isosurface using a chosen function applied to input vertex data. The input function for a specific use case creates a surface where the function has a chosen value.

An example use of an isosurface from a geometry shader is in the automated creation of natural looking terrain. Using appropriate combinations of functions to create the isosurface defines the real-time generated terrain that is produced.

9.2.1 How to visualize the creation of an isosurface

To understand how an effective isosurface is made from a function, it is helpful to start with a simple 2D example and then extrapolate the same ideas to a 3D example.

Several methods of visualizing isosurfaces exist, these include:

- Ray tracing.
- Generation of a polygonal mesh.

9.2.2 Surface nets

One well known method that creates a polygonal mesh is called marching cubes. However, using an alternative method called surface nets can produce a similar result with less geometry. This reduction in geometry reduces the required bandwidth. Surface nets are an effective method for Mali GPUs.

9.2.3 Related information

Polygonise (Paulbourke.net).

Basic 2D surface net approximation

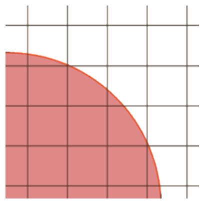
To start to understand surface nets start with a basic 2D surface net approximation.

A basic surface net creation method in 2D starts with the function overlaid on a grid. The function is negative one side of the surface, and positive the other side of the surface.

Check the four corners of each square in the grid to see if the function is positive or negative. If all corners have the same sign, then the cell is either completely inside or outside the surface. If there is a sign change between two corners, then the function intersects the edge between them.

The following figure shows a simple 2D isosurface:

Figure 9-2: A simple 2D isosurface.

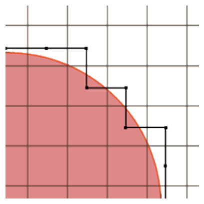


The red area represents the coordinates with a negative function value. The line between the red area and the white area is the isosurface.

To create a basic representation of the surface, place a vertex in the center of each square with an intersected edge and connect neighboring cells together.

The following figure shows how a basic approximation of the isosurface can be created.

Figure 9-3: A basic 2D surface net approximation.



9.2.4 Further information

Constrained Elastic SurfaceNets: Generating Smooth Models from Binary Segmented Data.

Smooth Voxel Terrain (Part 2).

9.2.5 Smoothed 2D surface nets

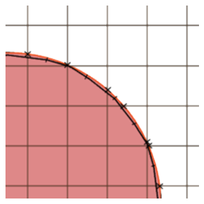
The basic 2D surface net approximation produces a rough result that requires smoothing to make it a closer fit to the real surface function.

One way of smoothing the result is to apply an iterative relaxation scheme that minimizes a measure of the global surface roughness. In each pass, this routine perturbs the vertices closer to the surface while keeping the point inside the original box. Keeping the points inside their original boxes ensures that the sharp features of the shape are preserved.

This process can be computationally expensive and hard to perform in real time. To reduce the computation time, take the average of the intersection coordinates for each. The result is a set of points that can be joined to make a good approximation of the true isosurface.

The following figure shows the intersection average approach.

Figure 9-4: A smoothed approximation of a 2D isosurface.

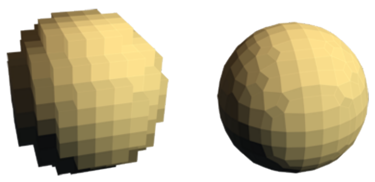


9.2.6 3D surface nets

The 2D surface net concept can be extended to 3D surfaces. Instead of considering squares, consider cubes. Instead of connecting points with lines, create planes.

The following figure shows an example basic polygon mesh for a 3D isosurface and the result of smoothing the same mesh.

Figure 9-5: A 3D approximation of an isosurface.



9.3 Implementing surface nets on the GPU

An example GPU implementation of surface nets uses multiple passes to create the isosurface.

- Sample each corner in the grid. Store the result of the potential function in a 3D texture.
- Fetch the function value at the eight corners of each cube, one cube at a time. Compute the average of the intersection points. Store the result in another 3D texture.
- For every cube that is on the surface, link neighboring cells on the surface together to produce faces. Passes one and two use compute shaders, and pass three uses a geometry shader.

9.3.1 Linking points in 3D

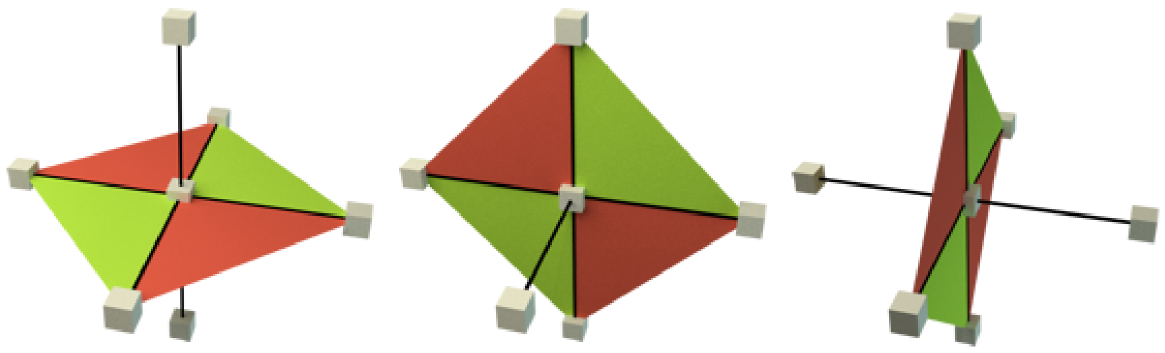
Linking the point on a 2D grid is easy to visualize. However, extending this linking concept to 3D can be harder to visualize.

Each cube on the surface can connect to its six neighbors using 12 possible triangles. A triangle is only created if both the neighbors being considered are on the surface. Creating triangles in this way across every cube in the grid creates redundant and overlapping triangles.

To avoid duplicating work, only construct triangles that point backwards from each cube. This means that three edges are considered for each cube. If one of the three edges exhibits a sign change, the vertices associated with the four cubes that contain the edge are joined to make a quad.

The following figure shows the possible faces that this method can produce:

Figure 9-6: Possible approximated faces for a 3D surface net.



9.3.2 Speeding up the geometry shader pass

You can optimize the geometry shader linking pass so that it runs faster. One way to increase the speed is to limit the number of cubes that the geometry shader considers.

Only process cubes that are known to be on the surface to reduce the computation required. Use indirect draw calls and atomic counters to enable the use of this optimization. This work starts during the previous compute shader stage.

If a surface intersects a cube, the compute shader writes the index for the cube to an index buffer and increments the count atomically. The following code shows an example implementation of this method:

```
        uint unique = atomicCounterIncrement(outCount);
        int index = texel.z * N * N + texel.y * N + texel.x;
        outIndices[unique] = uint(index);
```

In this code, N is the length of the cube sides in the grid.

When the code performs a draw call to the geometry shader, the following buffers can be bound to direct the geometry shader to operate on the correct cubes:

points_buffer: A buffer containing the grid of points.

index_buffer: The index buffer containing the points that require processing.

indirect_buffer: An indirect draw call buffer containing the draw call parameters.

The following example geometry shader functions use these buffers:

```
        glBindBuffer(GL_ARRAY_BUFFER, app->points_buffer);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, app->index_buffer);
        glBindBuffer(GL_DRAW_INDIRECT_BUFFER, app->indirect_buffer);
        glDrawElementsIndirect(GL_POINTS, GL_UNSIGNED_INT, 0);
```

9.4 An example use for geometry shaders

You can use geometry shaders to create a more realistic landscape for a scene. To do this, you must carefully choose the functions that create the terrain to produce the effect that you want.

9.4.1 Flat terrain starting point

A basic function produces a flat plane. The function is negative below the plane, and positive above it.

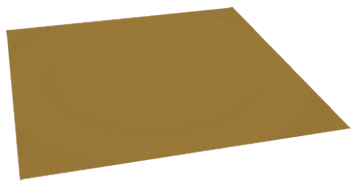
The following code shows how this surface can be described:

```
surface = p.y;
```

`p` is the sampling point in 3D space. surface is the function value that is stored in the 3D texture.

The following figure shows an example flat terrain:

Figure 9-7: An example flat terrain isosurface.



9.4.2 Oscillating terrain

Terrain is rarely completely flat. If you add an oscillating factor to the geometry shader function, it can produce more realistic terrain.

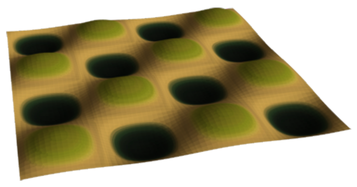
The following code shows how an oscillating surface can be defined:

```
        surface = p.y;
        surface += 0.1 * sin(p.x * 2.0 * pi) * sin(p.z * 2.0 * pi);
```

The result looks more realistic than flat terrain, but still does not resemble a real world landscape. This can be a useful starting point to build on.

The following figure shows an example oscillating terrain:

Figure 9-8: An example oscillating terrain isosurface.



9.4.3 The effect of adding noise to geometry shader isosurfaces

You can add noise to a surface function to make the terrain more realistic. This is a common technique in procedural modeling.

Noise types that can be added include:

- Simplex.
- Perlin.
- Worley.

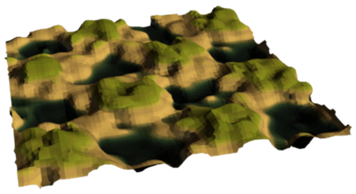
The following code shows an example implementation of noise in a geometry shader:

```
        surface = p.y;
        surface += 0.1 * sin(p.x * 2.0 * pi) * sin(p.z * 2.0 * pi);
        surface += 0.075 * snoise(p * 4.0);
```

This produces a more realistic effect than the base sine function without noise. However, you can add further functions to tweak the effect so that it suits the required use case better.

The following figure shows the effect of adding noise to an oscillating isosurface function:

Figure 9-9: The effect of adding noise to an oscillating isosurface function.



9.4.4 Flattening the tops of the terrain

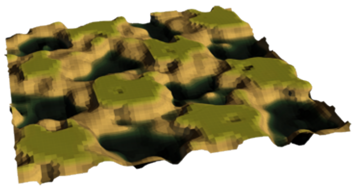
You can alter a geometry shader function so that the tops of the resulting hills in the terrain are flat.

The following code shows an example implementation of a flattening alteration:

```
        surface = p.y;
        surface += 0.1 * sin(p.x * 2.0 * pi) * sin(p.z * 2.0 * pi);
        surface += 0.075 * snoise(p * 4.0);
        surface = max(surface, -(p.y - 0.05));
```

The following figure shows the result of flattening the top of an isosurface function:

Figure 9-10: An example flattened top isosurface function.



9.4.5 Further reading

Distance functions.

9.4.6 How textures and shading can be applied

The mesh that geometry shaders produce can look unappealing without further processing. Adding textures and shading can dramatically improve the appearance of geometry.

You can create code that can add a different texture color for different heights in generated terrain. The result from this can be very effective. Using height with another value that determines a second coordinate in a texture can create variations in the texture at a set altitude. This effect looks more varied and natural than only using height.

One way that you can light the surface is to create code that approximates the gradient of the potential function in the geometry shader and normalize the result.

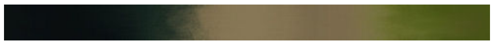
This method produces a faceted look because the normal is computer for each face separately. To create a smoother normal, approximate the gradient at each generated vertex and blend the results between them using the fragment shader.

The following code shows how this can be achieved:

```
        float v000 = texelFetch(inSurface, texel, 0).r;
        float v100 = texelFetch(inSurface, texel + ivec3(1, 0, 0), 0).r;
        float v010 = texelFetch(inSurface, texel + ivec3(0, 1, 0), 0).r;
        float v001 = texelFetch(inSurface, texel + ivec3(0, 0, 1), 0).r;
        n = normalize(vec3(v100 - v000,
        v010 - v000,
        v001 - v000));
```

The following figure shows an example texture that this method could use:

Figure 9-11: An example terrain texture.



Alternatively, you can create code to sample the gradient in the fragment shader. However, this is more computationally expensive.

10 Tessellation

This chapter details some of the pros and cons of tessellation, common tessellation methods, ways to optimize tessellation if it is right for your project, what adaptive tessellation is, aliasing, and MipMap selection.

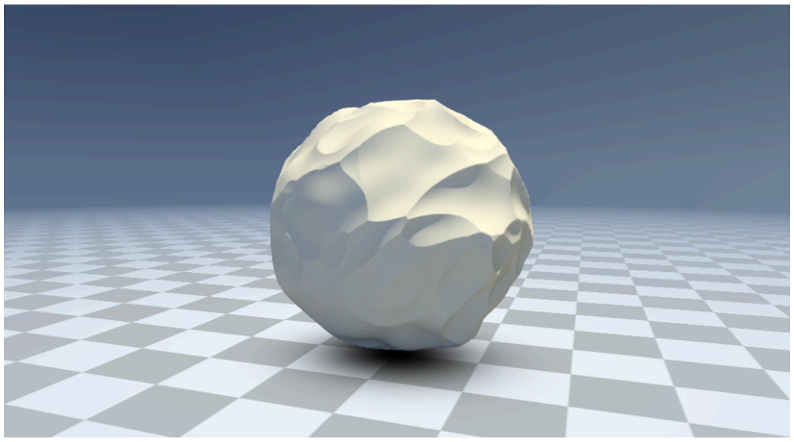
10.1 About tessellation

Tessellation is a graphics technique that creates geometry using your Mali GPU. You can adjust the amount of new geometry that is created and the shape that the new geometry takes. Tessellation operates on each vertex individually, and can operate on all the vertices of required primitives.

The ability to use tessellation on an Android device with a Mali GPU is included in the *Android Extension Pack* (AEP).

The following figure shows an example of the type of result that tessellation can achieve:

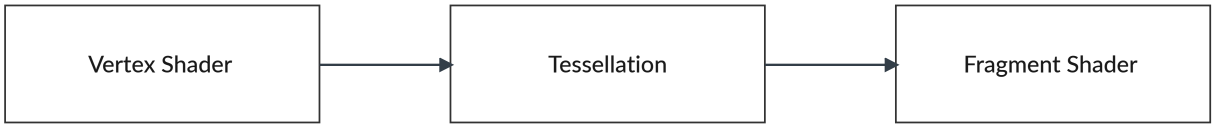
Figure 10-1: An example result of tessellation.



Tessellation adds some optional graphics pipeline stages between the vertex shader and fragment shader.

The following figure shows the position of tessellation in the graphics pipeline:

Figure 10-2: Tessellation pipeline position.



The ability to create geometry on the GPU is useful. However, there are some common problems that you must consider when you use tessellation. These include:

10.1.1 Performance problems

It is possible to create too many triangles that are too small to be seen or be effective. The computation time that is required to create these triangles can cause performance problems and is unnecessary.

10.1.2 Patch gaps

Patch gaps occur when:

- Tessellation is not applied when it supposed to be applied.
- The tessellation factors for adjacent edges do not match causing different levels of patch subdivision.
- The texture filtering that samples the displacement map accesses different texels for neighboring edges. This can happen if a displacement map texture is wrapped around an object. Where two edges of the texture meet, texel sampling can sample both sides of the texture producing unintended results.

10.1.3 Mesh swimming

This is where vertices appear to move around as the camera moves closer to a tessellated object. This normally happens when the tessellation factors that subdivide the new primitives are dynamically set using a metric of some kind.

10.1.4 How tessellation works

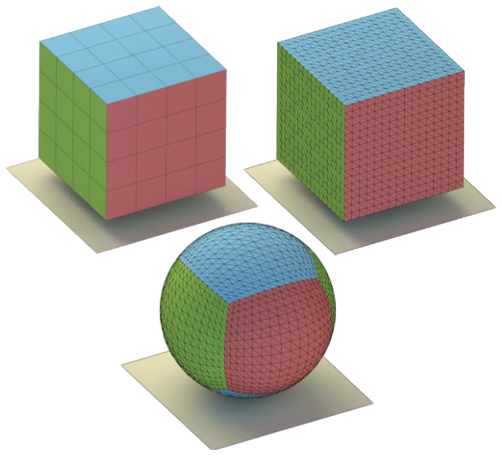
Tessellation splits primitives up into a larger number of smaller primitives. It then moves the new vertices to create more complex geometry than existed originally.

The number of times that primitives are divided, and the way the new vertices are moved are programmable. A tessellation control shader controls the subdivision of primitives, and a tessellation evaluation shader controls the movement of the new vertices.

The initial subdivision of an object surface creates a mesh made from quads or triangles.

The following figure shows an example subdivision to create the mesh, and the result of a basic displacement that creates a ball:

Figure 10-3: An example tessellation process.



10.2 Common tessellation methods

There are several methods that you can use to apply tessellation. You can combine these methods to create a different overall effect. Which method or methods work best depends on the situation and outcome you require.

10.2.1 Displacement mapping

Artists create models that have significantly higher triangle counts before they are placed into a game. When they are put into the game, they must have their triangle counts reduced to meet the available polygon budget. Displacement mapping stores this information so that it can be restored at runtime.

Displacement mapping stores the details for the high-quality mesh as a texture. Applying an LOD scheme to this method avoids unnecessary work.

10.2.2 Subdivision surfaces

Subdivision surfaces are used to create smooth surfaces from a less smooth, less detailed input.

This method refines an input polygonal mesh towards the recursive limit of the process being used. This creates a good approximation of the intended smooth surface.

10.2.3 Smoothed 2D surfaces

GUI elements and text can be described using higher-order geometry such as Bezier curves. Using these techniques it is possible to produce infinitely smooth curves. You can use tessellation to generate the geometry to render smooth 2D graphics.

10.2.4 Continuous Level Of Detail (LOD)

Geometry that covers a small number of pixels requires less tessellation work than geometry that covers more pixels. One method that uses this property to its advantage is called continuous Level Of Detail (LOD). This technique avoids creating extra triangles where they are not visible or obvious.

Continuous LOD algorithms provide an apparently continuous transition between detail levels. Discrete LOD algorithms perform a similar job, but can produce popping when the algorithm switches between discrete detail levels.

Continuous LOD uses displacement mapping and smoothing.

You can combine discrete LOD and continuous LOD to create the effect you want.

10.3 Optimizations for tessellation

There are several optimizations that you can apply to tessellation methods to improve performance.

Optimizations discard patches before tessellation is performed, depending on whether a patch is visible or not.

10.3.1 Backface culling

Some of the vertices that tessellation creates are invisible in the final image that is rendered to the screen. Removing patches before they are sent to the tessellation control shader avoids unnecessary GPU work.

If a patch is not going to be seen, then setting the tessellation factors for the patch to zero culls it. This stops any additional geometry from being created using that patch.

If there is already a GPU backface culling pass in your graphics pipeline, it is performed after the tessellation process. A separate pass is required during the tessellation process for it to benefit tessellation.

The control shader can check whether a patch is visible in the final rendering. To perform this check, you must program the control shader to examine each vertex of a patch to determine if it is facing away from the camera. The result for a perfect sphere is that patches with all their vertex normals facing away from the camera are hidden.

However, checking if a patch faces towards the camera or away can cause some problems. Some patches that do not face the camera change when they are tessellated so that they reach far enough out from their original position to be visible.

Culling these patches because they face away from the camera means that when they turn towards the camera, the tessellated effect appears suddenly. This does not produce a good result. To reduce the impact from this problem, set the culling stage to cull patches with a range of normal values that still includes patches that face backwards near the edge. You can tune this range to suit the use case.

10.3.2 Occlusion culling

Occlusion culling tests whether an object is behind another object. If an object is hidden, then occlusion culling prevents it from being tessellated.

You can implement occlusion culling per-object and per-patch. You can create algorithms that test whether an object occludes part of itself, which can be useful for more complex geometry.

Some patches that are behind other objects before tessellation can grow to become visible after tessellation. To reduce the impact from the problem, allow patches that are behind another object by less than the maximum tessellation displacement to remain, without culling them.

10.3.3 Frustum culling

Frustum culling tests whether an object is in the camera view. If an object is outside the view of the camera, then tessellation that is performed on it is not visible and can be avoided.

Frustum culling can be performed on the application processor or your Mali GPU. Frustum culling on the application processor is performed per-object, and frustum culling on your Mali GPU is performed per-patch.

Use the control shader to check whether a patch is inside the view area. If a patch is on some geometry that is behind the camera, then tessellating it is unnecessary.

To check whether a patch is inside the view area, the shader must project the patch vertices onto normalized device coordinates and compare the result with the frustum bounds. If every vertex for a patch is completely off screen, then it can be culled.

If there is already a GPU frustum culling pass in your graphics pipeline, it is performed after the tessellation process. A separate pass is required during the tessellation process for it to benefit tessellation.

You can use frustum culling on the application processor and GPU together to increase performance.

10.3.4 Application processor frustum culling

Frustum culling is normally performed on the application processor. Test each object to see if the bounding box that defines it is within the camera view. If an object bounding box is outside the view, then do not tessellate it.

To avoid culling objects that grow into the camera view without culling enabled, ensure that you increase the size of the bounding box to allow for this.

10.3.5 GPU frustum culling

Frustum culling on your Mali GPU is similar to frustum culling on the application processor. However, instead of testing each object, test each patch.

Carefully adjust the patch bounding box that the GPU frustum culling uses to avoid popping effects that can occur when an object moves into the screen space and becomes tessellated.

The AEP PrimitiveBoundingBox extension defines a variable called `gl_BoundingBox`. This extension enables frustum culling in the tessellation control shader using the GPU. To use `gl_BoundingBox`, fill it with the bounding box info for the patch you are checking.

10.4 Adaptive tessellation

You can use several adaptive tessellation techniques to prevent unnecessary details being produced that cannot be seen.

Adaptive tessellation techniques reduce the amount of geometry that tessellation creates, depending on the position of patches in the camera view.

10.4.1 Progressive level of detail

Some patches require less tessellation than others. For example, a patch requires less detail when it is a long way away from the camera than when it is close to it. Progressive level of detail changes the tessellation factor to minimize tessellation where it is not noticeable and increases it where it is most noticeable.

The camera can move so that the amount of tessellation a patch requires changes. This means that your code must reassess the required tessellation factor regularly.

Some metrics that you can use to determine how much tessellation is required are:

- The distance between each patch and the camera.
- The angle that each patch is being viewed from.
- Screen space that each patch occupies.

10.4.2 Distance-based tessellation

Applying a high level of tessellation works well for objects close to the camera where the most detail is visible. However, objects that are further away from the camera occupy a smaller area of the screen. Tessellating these objects to the same level as closer objects creates a large amount of detail that cannot be seen because it is too small. Distance-based tessellation works to reduce this problem.

To implement this technique, you must create code that sets a maximum tessellation factor for each edge using the distance from the camera to the patch being processed. Adjust the reduction so that close objects look highly detailed, but far objects are tessellated less or not tessellated.

10.4.3 View-angle based tessellation

If an object is facing the camera, then normal maps are usually sufficient for most of the side of the object that faces the camera. However, the edges of the object benefit from tessellation, because they are seen in profile.

Using tessellation along the edges of objects and reducing its use in other places helps minimize the performance impact of tessellation, and maximize its effectiveness.

10.4.4 Screen-space tessellation

If a patch occupies a large area of the screen, then it requires more tessellation than a patch that occupies a small area of the screen. Screen-space tessellation adjusts the tessellation factor to use this characteristic.

Distance-based tessellation and view-angle based tessellation do not stop small patches near the screen from being tessellated too much. Without screen-space tessellation, these patches produce triangles that are too small to see.

Create code that analyzes the size of a patch onscreen and sets the tessellation factor using this information.

10.4.5 Frequency-based tessellation

Areas that have high frequency details benefit from tessellation more than areas that are smoother. Frequency-based tessellation adjusts the tessellation factor using a map that indicates the location of high frequency detail in the initial 3D model.

To use frequency-based tessellation, create a frequency map from the initial 3D model offline, and send it to the tessellation control shader at runtime. Use this map to set the tessellation factor for each patch.

You can extend this analysis further. If a large area has a similar displacement, then you can displace it instead of tessellating it.

10.4.6 Geometry adaptive tessellation

Geometry adaptive tessellation examines the curvature of the surface before tessellation is applied. If a section of terrain or an object is flat, then tessellation is sometimes not as important as it is for strongly curved areas. For example, a carpet does not usually benefit from tessellation.

However, an object that starts flat but has a highly detailed displacement map might still require a high tessellation factor.

10.4.7 Combining adaptive tessellation techniques

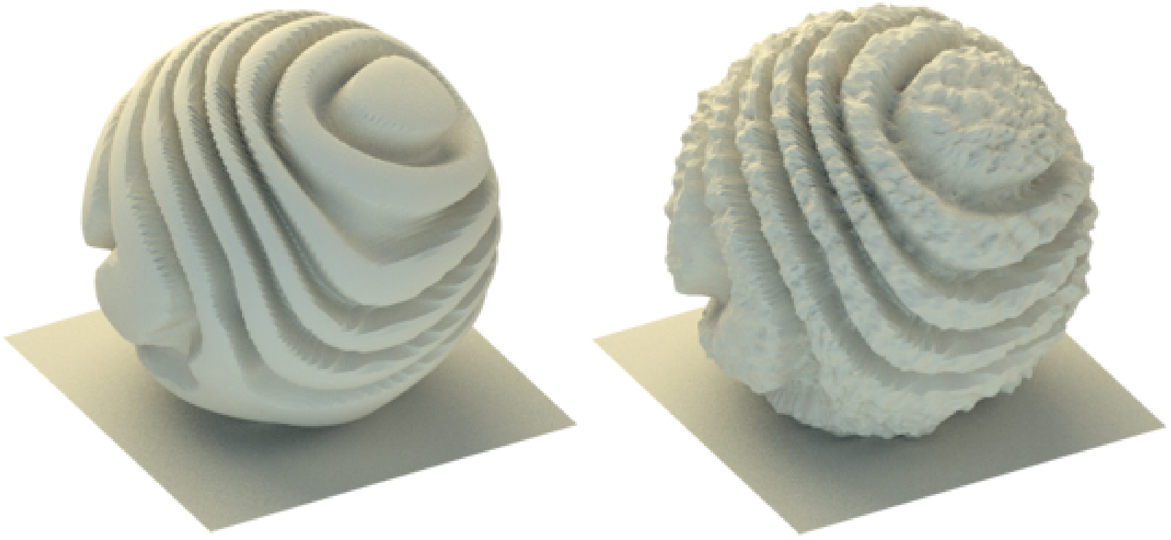
You can combine different adaptive tessellation techniques to maximize the effect. To do this, create a weighted average of the result of the different analysis stages and adjust the tessellation value using it. You can vary the weighting of the average to produce the effect you want.

10.5 Aliasing

If a displacement map has too much detail for the sample rate of the tessellation mesh, then aliasing can occur. This reduces the perceived quality of the result. Some alterations can be made to avoid this problem.

Adding noise can disguise aliasing. The following figure shows the effect of noise on a tessellated sphere displaying aliasing.

Figure 10-4: An example of aliasing.



A technique called importance sampling can reduce aliasing. This technique moves tessellation points so that they align more closely with the contours of the displacement map.

10.6 Mipmap selection

If the sampling rate is too low, then sudden variations can appear when the tessellation factor increases. Adjusting the frequency of the displacement map can improve this problem.

Using different precalculated mipmaps for the displacement map can reduce the effect of this problem. For example, if a low tessellation factor is being used, the sample rate of the displacement map is low. Use a lower mipmap level to ensure that high frequency detail in the displacement map does not affect the result.

Create code that analyzes the frequency of components in the displacement map and the tessellation level, to enable the selection of the best mipmap level for a specific situation. The result of this analysis must be passed to the tessellation evaluation shader.

11 Android Extension Pack

Android Extension Pack enhances OpenGL ES 3.1 gaming with a number of extensions.

11.1 Using Android Extension Pack

The extensions in AEP are treated as a single package by Android OS. They are exposed in a single extension.

If the following extension is present, your application can assume all the extensions in the package are present: `ANDROID_extension_pack_es31a`

You can enable the shading language features with the following statement: `#extension`.

Note

AEP requires OpenGL ES 3.1.

In your application manifest, you can declare that your application must be installed only on devices that support AEP:

```
<manifest>
        <uses-feature
                android:name="android.hardware.opengles.aep"
                android:required="true" />
        ...
</manifest>
```

11.2 Android Extension Pack extensions

AEP includes the following features and extensions:

ASTC texture compression: `KHR_texture_compression_astc_ldr`.

Debugging features: `KHR_debug`.

Fragment Shaders: Guaranteed support for shader storage buffers, images, and atomics.

Framebuffer operations: `EXT_draw_buffers_indexed`, `KHR_blend_equation_advanced`.

Per-sample processing: `OES_sample_shading`, `OES_sample_variables`, `OES_shader_multisample_interpolation`.

Texturing functionality: `EXT_texture_cube_map_array`, `EXT_texture_sRGB_decode`, `EXT_texture_buffer`, `EXT_texture_border_clamp`, `OES_texture_stencil8`, `OES_texture_storage_multisample_2d_array`.

Tessellation and Geometry: `EXT_tessellation_shader`, `EXT_geometry_shader`, `EXT_shader_io_blocks`, `EXT_primitive_bounding_box`.

Appendix A Revisions

This appendix provides additional information that is related to this guide.

Table A-1: Version 1.0

Change	Location	Affects
First Release	–	First release of v1.0

Table A-2: Version 1.1

Change	Location	Affects
Added information about Pixel Local Storage.	Pixel Local Storage	First Release of v1.1
Added information about Framebuffer Fetch.	Framebuffer Fetch	First Release of v1.1
Updated Mali GPU Hardware Families page.	About the Mali GPU hardware families	First Release of v1.1
Updated ASTC chapter.	Adaptive Scalable Texture Compression ASTC texture compression	First Release of v1.1