



Arm[®] Mali[™] GPU OpenGL ES 3.x

Version 1.1

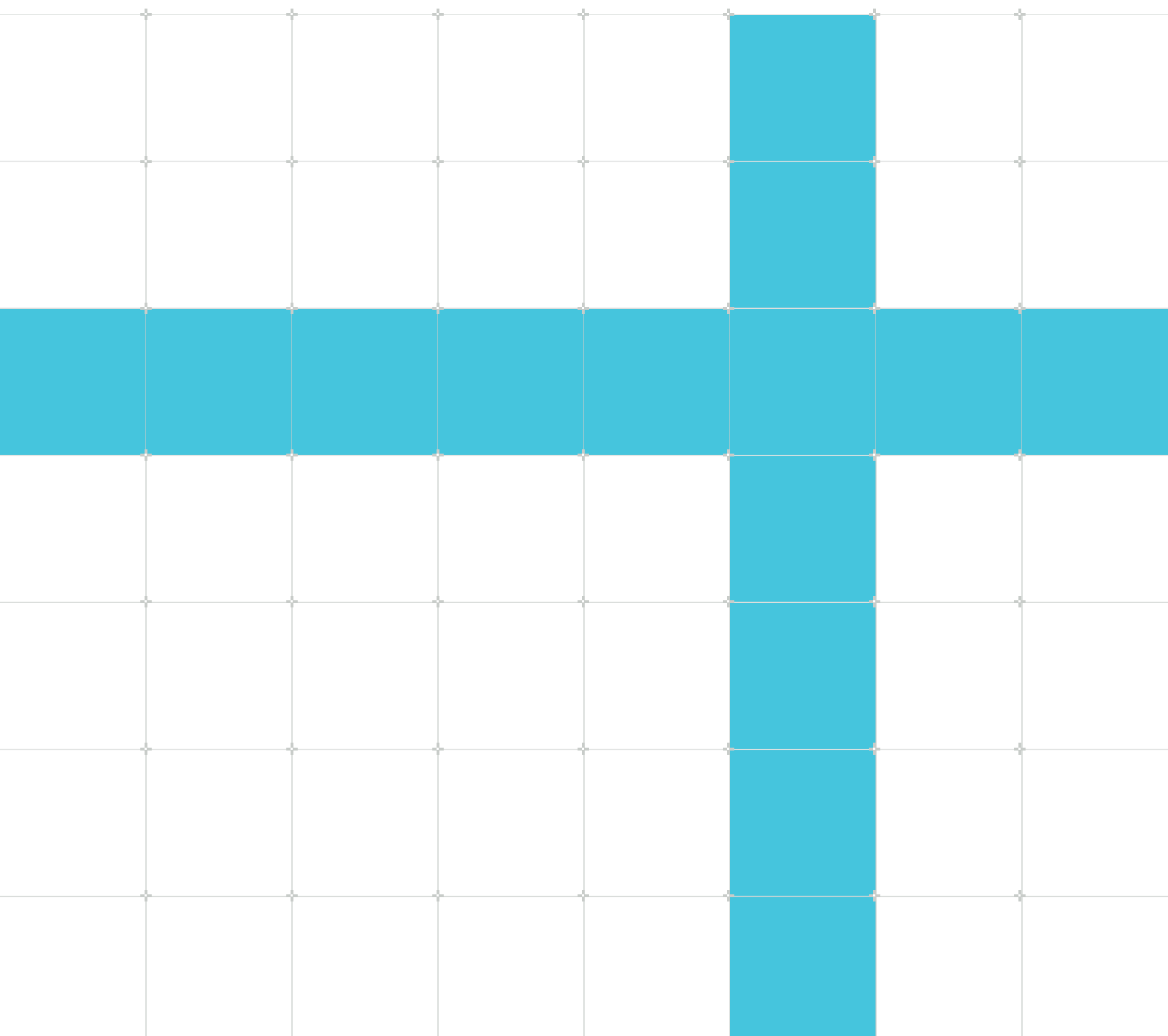
Developer Guide

Non-Confidential

Copyright © 2016, 2021, 2023 Arm Limited (or its affiliates).
All rights reserved.

Issue 01

100587_0101_01_en



Arm® Mali™ GPU OpenGL ES 3.x

Developer Guide

Copyright © 2016, 2021, 2023 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-00	11 March 2016	Non-Confidential	Initial release
0101-00	30 July 2021	Non-Confidential	First release of version 1.1
0101-01	27 June 2023	Non-Confidential	Provided clarity to our Compute Shaders Example

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND

REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2016, 2021, 2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

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.....	7
1.1 Conventions.....	7
1.2 Useful resources.....	8
1.3 Other information.....	8
2. Introduction to OpenGL ES 3.x on Mali GPUs.....	9
2.1 About OpenGL ES 3.0.....	9
2.2 About OpenGL ES 3.1.....	10
2.3 About the Android Extension Pack.....	11
2.4 About OpenGL ES 3.2.....	11
3. The Mali GPU Hardware Families.....	12
3.1 About the Mali GPU hardware families.....	12
3.2 About the Mali GPU hardware features.....	12
3.3 Valhall architecture hardware components.....	13
4. The OpenGL ES graphics pipeline.....	15
4.1 About the OpenGL ES 3.x pipeline.....	15
4.2 Primitive assembly.....	16
4.3 Vertex processing.....	17
4.4 Rasterization.....	17
4.5 Fragment processing.....	17
4.6 Framebuffer operations.....	18
5. Pixel Local Storage.....	19
5.1 About Pixel Local Storage.....	19
5.2 Using Pixel Local Storage.....	21
5.3 Examples of Pixel Local Storage.....	23
6. Related extensions.....	26
6.1 About related extensions.....	26
7. ASTC texture compression.....	28
7.1 About textures.....	28

7.2 Texture requirements.....	28
7.3 Limitations of existing texture compression formats.....	29
7.4 ASTC features and quality.....	29
7.5 2D texture compression.....	30
7.6 3D texture compression.....	31
7.7 ASTC types and extensions.....	32
7.8 The astcenc application.....	33
8. Compute shaders.....	39
8.1 About compute shaders.....	39
8.2 Work groups.....	40
8.3 Compute shaders example.....	41
9. Geometry shaders.....	45
9.1 About geometry shaders.....	45
9.2 Isosurfaces.....	46
9.3 Implementing surface nets on the GPU.....	48
9.4 An example use for geometry shaders.....	50
10. Tessellation.....	54
10.1 About tessellation.....	54
10.2 Common tessellation methods.....	56
10.3 Optimizations for tessellation.....	57
10.4 Adaptive tessellation.....	59
10.5 Aliasing.....	60
10.6 Mipmap selection.....	61
11. Android Extension Pack.....	62
11.1 Android Extension Pack extensions.....	62

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
<i>italic</i>	Citations.
bold	Interface elements, such as menu 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 <u>underline</u>	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: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . 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.
 Note	An important piece of information that needs your attention.

Convention	Use
 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 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Non-Arm resources	Document ID	Organization
<i>Dist Functions</i>	–	Dist Functions
<i>Geometry</i>	–	Geometry
<i>Polygonise</i>	–	Polygonise
<i>Smooth Voxel Terrain Part2</i>	–	Smooth Voxel Terrain Part2



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 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.



OpenGL ES 3.0 is backwards compatible with OpenGL ES 2.0.

2.2 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.



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

2.3 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.



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.

2.4 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

There are families of Mali GPUs: the Utgard architecture family, and the Midgard architecture family.

The Midgard architecture family

The Midgard architecture family of Mali GPUs have unified shader cores that perform vertex, fragment, and compute processing. The Midgard architecture Mali GPUs support OpenGL ES versions 1.1, 2.0, 3.0, 3.1, 3.2, and Vulkan. They also support compute applications with OpenCL 1.1, 1.2 and Renderscript.

The Utgard architecture family

The Utgard architecture family of Mali GPUs have a vertex processor and one or more fragment processors. They are used for graphics-only applications with OpenGL ES 1.1 and 2.0.



AEP and OpenGL ES 3.0 to 3.2 do not work on Utgard GPUs.

3.2 About the Mali GPU hardware features

Mali GPUs contain the following common hardware:

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.

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.



Mali GPUs use L2 cache in place of local memory.

3.3 Valhall architecture hardware components

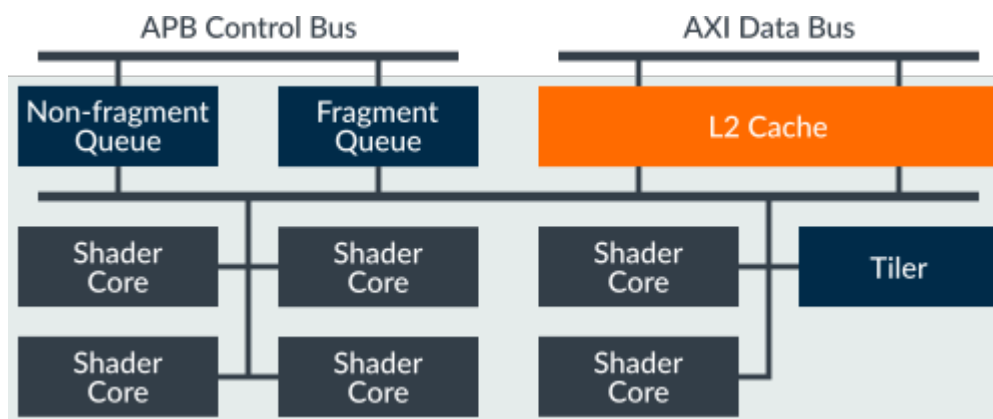
The Valhall family of Mali GPUs uses the same top-level architecture as the previous generation Bifrost GPUs. The Valhall family uses a unified shader core architecture.

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 3-1: 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

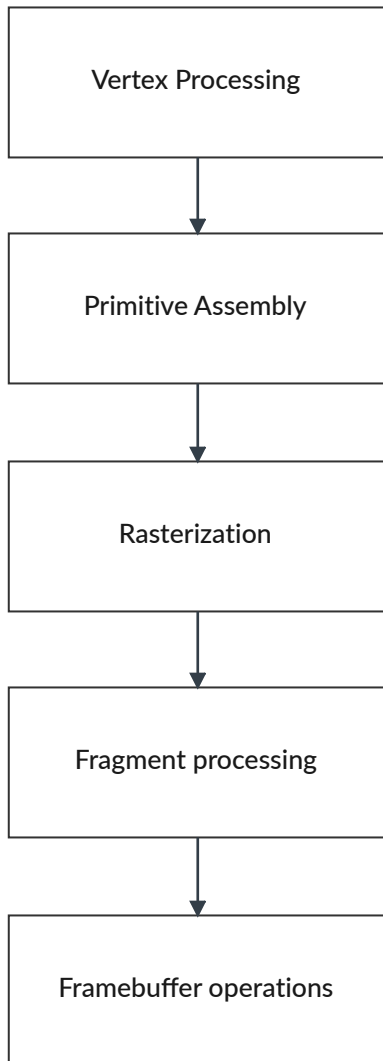
Mali GPUs use data structures and hardware functional blocks to implement the OpenGL ES graphics pipeline.

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.

Reads data

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

Rasterizes polygons

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

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.

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

Pixel Local Storage (PLS) is an extension for OpenGL ES 3.0 or higher that enables you to temporarily store data in a buffer on the GPU.

Advantages of PLS

PLS saves memory bandwidth usage and increases performance.

Mali GPUs render images in 16 pixel by 16 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 possible on mobile devices.

Operation of PLS

PLS provides 16 bytes 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.

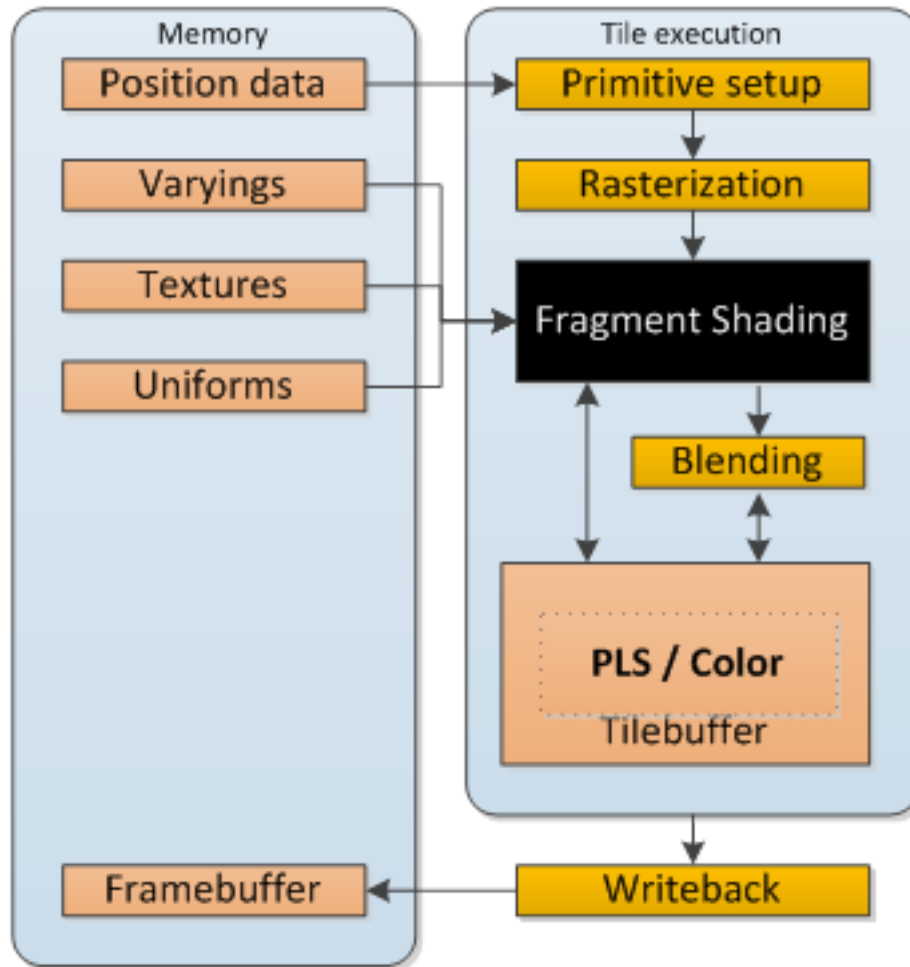
To generate output, each of your shaders builds up information in turn in the PLS. You create a final custom shader known as a resolve shader, to generate the color information, and then write it to the framebuffer in the correct format.

PLS rendering pipeline changes

Enabling PLS modifies the graphics rendering pipeline.

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

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



There are two paths to the PLS storage:

- One provides read and write access directly from the fragment shader.
- The other lets you write color values through the fixed function blender.

The path through the blender is required by the final stage that writes the color to the buffer. Writing through the blender converts the data to the native framebuffer format when it is written back to main memory.

5.2 Using Pixel Local Storage

This section describes using Pixel Local Storage.

Enabling Pixel Local Storage

You must enable Pixel Local Storage at the beginning of the frame lifetime and disable it at the end of the frame lifetime.

PLS is exposed as the extension `EXT_shader_pixel_local_storage`.

Use `GL_SHADER_PIXEL_LOCAL_STORAGE` to enable and disable PLS. You must do this on a framebuffer basis.

You must disable PLS before switching framebuffer.

During the framebuffer lifetime, draw calls interact with PLS to build up the final pixel color.

PLS interface qualifiers

The Pixel Local Storage extension includes a set of interface qualifiers that you use to declare the PLS variables. These describe how PLS variables are accessed between fragment shader invocations.

The following table lists the interface qualifiers:

Table 5-1: PLS interface qualifiers

Interface	Description
<code>__pixel_localEXT</code>	Storage can be read from and written to.
<code>__pixel_local_inEXT</code>	Storage can be read from.
<code>__pixel_local_outEXT</code>	Storage can be written to.



You can use `__pixel_local_inEXT` and `__pixel_local_outEXT` to reinterpret data between shader invocations.

PLS layout qualifiers

The Pixel Local Storage extension includes a set of layout qualifiers that you use to declare the PLS variables. These describe how data is stored in the buffer.

The following table lists the layout qualifiers:

Table 2-2 PLS layout qualifiers

Layout	Base Type
<code>r32ui</code>	<code>uint</code> , default choice
<code>r11f_g11f_b10f</code>	<code>vec3</code>

Layout	Base Type
r32f	float
rg16f	vec2
rgb10_a2	vec4
rgba8	vec4
rg16	vec2
rgba8i	ivec2
rg16i	ivec2
rgb10_a2ui	uvec4
rgba8ui	uvec4
rg16ui	uvec2

The following code is an example of a PLS interface block:

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



The compiler requires that the vector sizes of the individual PLS variables match the number of components in the layout qualifier.

For more information see the *shader pixel local storage* extension specification at: [Khronos.org](https://www.khronos.org).

Caveats to using PLS

Be aware of the following caveats when you are using Pixel Local Storage:

- Clearing PLS is not trivial:
 - You can clear to 0, this effectively clears all the PLS variables to 0.
 - You can clear individual PLS variables using a full-screen quad.
- Blending is not permitted when you are using PLS. You must either do blending manually, or by first resolving PLS and then doing alpha blending using the fixed function blender.
- Multi-sampling and multiple render targets are not supported.

Performance considerations when using PLS

Be aware of the following performance considerations when you are using Pixel Local Storage:

- Avoid GPU pipeline bubbles:
 - Reading from PLS can cause GPU pipeline bubbles. You can work around this by scheduling the PLS read as late as possible in your shader.
 - Avoid using short sequential shaders that read from PLS.

- Store depth in a PLS variable:
 - Reading depth using `shader_framebuffer_fetch_depth_stencil` causes bigger GPU pipeline disruptions than using PLS.
 - If you are required to read depth often, storing the depth in PLS produces better performance.
 - Storing depth in a PLS variable is useful for techniques such as soft particles and deferred shading.

Alternatives to PLS

You can get some of the same benefits as Pixel Local Storage using a different technique.

You can get some of the same benefits as PLS by using *Multiple Render Targets* (MRT) together with the extension `GL_EXT_shader_framebuffer_fetch`, but there are some things to consider first:

- This technique only works if you can ensure that the driver knows not to write the render targets back to main memory.
- PLS is more explicit than MRT. It is harder for your application to get it wrong, and the driver does not have to try to work out if a render target is used or not.
- PLS is also more flexible. Your shaders can reinterpret data between shader invocations, and it is not limited to the OpenGL ES 3.x framebuffer formats.

5.3 Examples of Pixel Local Storage

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

Deferred shading

This section describes an implementation of deferred shading using Pixel Local Storage: A typical implementation of deferred shading using Pixel Local Storage involves splitting the rendering into three passes:

1. G-buffer generation pass.
2. Shading pass.
3. Combination pass.

G-buffer generation pass

In this pass, the diffuse color and normal properties for each pixel are calculated.

Instead of writing these to normal color outputs, they are placed into the PLS store.

In this example the fragment shader declares a PLS output block and writes to it:

```
_pixel_local_outEXT FragData
{
    layout(rgba8) highp vec4 Color;
    layout(rgl6f) 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;
}

```



Deferred shading is easier with PLS than MRT, because you do not have to explicitly allocate the g-buffer targets.

Figure 5-2: G-buffer generation pass.



Shading pass

In the shading pass, the properties are read from the PLS store and lighting values are calculated based on them. The calculated values are accumulated in the PLS:

```

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

```

Figure 5-3: Shading pass.



Combination pass

The values in the PLS store are used to calculate the final value of the pixel:

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

out highp vec4 fragColor;

void main()
{
    fragColor = resolve(gbuf.Color,
                       gbuf.LightingRG.x,
                       gbuf.LightingRG.y,
                       gbuf.NormalZ_LightingB.y);
}
```

Figure 5-4: Combination pass.



6. Related extensions

This chapter describes related extensions.

6.1 About related extensions

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

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
```

Variable for **GL_ARM_shader_framebuffer_fetch** extension

- Variable: `gl_LastFragColorARM`
- Type: `vec4`
- Description: Reads the existing framebuffer color.

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
```

Variables for **GL_ARM_shader_framebuffer_fetch_depth_stencil** extensions

- Variable: `gl_LastFragDepthARM`
- Type: `float`
- Description: Reads the existing framebuffer depth value.



Value is in window coordinate space.

-
- Variable: `gl_LastFragStencilARM`
 - Type: `int`

- Description: Reads the existing framebuffer stencil value.

7. ASTC texture compression

This chapter describes *Adaptive Scalable Texture Compression* (ASTC).

7.1 About textures

Textures describe the surface properties of objects.

In 3D graphics, the geometry describes the shape and location of objects. Shaders use textures to give flat geometry the appearance of surface detail.

Shaders can color the surfaces of the object as well as add lighting, surface detail, reflectivity, and other properties. The textures used can be large, often several megabytes each. These have to be read every time the shaders run.

In a mobile device, bandwidth uses a lot of power so reducing it is important for both performance and battery life. Using texture compression saves memory bandwidth.

7.2 Texture requirements

There are many different uses for textures and these have different requirements.

Some types of textures require different numbers of color components:

- Reflectance textures:
 - Diffuse color requires three color components.
 - Diffuse color with alpha requires four color components.
- Gloss maps require one color component.
- Height maps require one color component.
- Normal maps:
 - Require three color components.
 - You can use a two component map and compute the third component in a shader.

Different uses of textures also have different requirements for quality:

- Reflectance textures only ever have a value between 0 and 1 so they can use *Low Dynamic Range* (LDR) textures.
- Diffuse map textures tend to require high-resolution textures.
- Lighting textures can be highly variable so these require floating-point or *High Dynamic Range* (HDR) textures.
- Lightmaps and environment maps typically require low-resolution textures.

- Gloss maps require high-quality textures.
- Normal maps require high-quality textures.
- User Interfaces require very high-quality textures.

3D textures are possible but are not used very often:

- They are likely to use a large amount of memory.
- There are very few compression formats available.

7.3 Limitations of existing texture compression formats

There were other texture compression formats available before ASTC.

The other texture compression formats have a number of limitations:

- Most texture formats only support a relatively small number of texturing requirements and quality levels.
- There is not much support for one or two channel images.
- There are not many options for trading off size against quality.
- None of the compression formats are cross platform so you must use different formats on different platforms.
- Some texture formats are proprietary so you can only use these on devices from particular vendors.

7.4 ASTC features and quality

Adaptive Scalable Texture Compression (ASTC) is a technology developed by Arm and AMD that has been adopted as an official extension to both the Open GL and OpenGL ES graphics APIs. ASTC produces high quality compressed textures at low bit rates and in many different types of image format configurations.

ASTC has the following features:

- ASTC is an OpenGL and OpenGL ES approved extension. Supports a large number of formats.
- ASTC is flexible, featuring bit rates from eight *bits per pixel* (bpp) down to less than one bpp. This enables you to fine-tune the tradeoff of space against quality.
- Supports from one to four color channels, together with modes for uncorrelated channels for use in mask textures and normal maps.
- Supports both LDR and HDR images.
- Supports both 2D and 3D images.
- ASTC is not proprietary, you can use ASTC on any platform that supports it.
- For the first time it is possible to use 3D textures to store, for example, discrete 3D object info.

Even with all this flexibility, quality is better than existing texture compression formats for LDR images, and is comparable to other formats for HDR.

All of these features are interoperable. You can choose any combination that suits your requirements.

The variety of block sizes gives you a lot of choices in how to compress a texture. For example, if an object is far from camera you can use a higher compression ratio because the visual quality is not important.

Alternatively, if the object is closer to camera the image quality is more important. You can use a lower compression ratio to produce better quality textures.

7.5 2D texture compression

Adaptive Scalable Texture Compression (ASTC) supports a wide range of formats and compression ratios for 2D texture compression.

ASTC compresses images in blocks of 128-bits. Each 128-bit block holds data for a number of pixels at different numbers of bits per pixel.

ASTC supports the following 2D formats:

Table 7-1: Bitrates of raw format images

Raw input format	Raw input bits per pixel
HDR RGB+A	64
HDR RGBA	64
HDR RGB	48
HDR XY+Z	32
HDR X+Y	32
RGB+A	32
RGBA	32
XY+Z	24
RGB	24
HDR L	16
X+Y	16
LA	16
L	8

You can choose between a range of compression ratios and quality levels for 2D textures:

Table 7-2: 2D compression bit rates

Block size	Bits per pixel
4 x 4	8.0

Block size	Bits per pixel
5 x 4	6.4
5 x 5	5.12
6 x 5	4.27
6 x 6	3.56
8 x 5	3.2
8 x 6	2.67
10 x 5	2.56
10 x 6	2.13
8 x 8	2.0
10 x 8	1.6
10 x 10	1.28
12 x 10	1.07
12 x 12	0.89

For RGBA, 8 bit per channel textures, the different block sizes give you these compression ratios:

Table 7-3: Compression ratios for RGBA, 8 bit per channel textures

Block size	Bits per pixel
4 x 4	4.0
5 x 4	5.0
5 x 5	6.25
6 x 5	7.50
6 x 6	9.00
8 x 5	10.01
8 x 6	12.01
10 x 5	12.53
10 x 6	15.00
8 x 8	16.00
10 x 8	20.08
10 x 10	24.97
12 x 10	30.12
12 x 12	35.93

7.6 3D texture compression

Adaptive Scalable Texture Compression (ASTC) supports a wide range of formats and compression ratios for 3D texture compression.

You can store a 256x256x256 arbitrary vector field in less than 300KB.

There are many possible uses for 3D textures, such as:

- Smoke.
- Light fields.
- Particle flow.
- Deep surface texture.
- Scene parameters.

ASTC compresses images in blocks of 128-bits. Each 128-bit block holds data for a number of pixels at different numbers of bits per pixel.

You can choose between a range of compression ratios and quality levels for 3D textures:

Table 7-4: 3D compression bit rates

Block size	Bits per pixel
3 x 3 x 3	4.74
4 x 3 x 3	3.56
4 x 4 x 3	2.67
4 x 4 x 4	2.00
5 x 4 x 4	1.6
5 x 5 x 4	1.28
5 x 5 x 5	1.02
6 x 5 x 5	0.85
6 x 6 x 5	0.71
6 x 6 x 6	0.59

7.7 ASTC types and extensions

There are a number of ASTC extensions supported in OpenGL ES:

- `GL_KHR_texture_compression_astc_ldr`: Supports 2D textures with LDR. Hardware vendors can decide to implement just the LDR specification.
- `GL_KHR_texture_compression_astc_hdr`: Extends the LDR profile to add HDR support.
- `GL_OES_texture_compression_astc`: Adds support for compressed 3D textures. This is optional in some GPUs.



The *ASTC Full Profile* is defined by these three extensions combined.

The Mali GPUs, including the Mali-T620, Mali-T720, Mali-T760, Mali-T820, Mali-T830, Mali-T860, and Mali-T880 support ASTC full profile.

7.8 The *astcenc* application

Arm provides an evaluation application called *astcenc* that can compress textures with ASTC.

astcenc basic options

The *astcenc* application runs from the command line and provides a number of basic options.

Compressing a texture

The *astcenc* application provides an option to compress a texture.

To compress a texture, use the *astcenc* command with the `-c` option:

```
astcenc -c [options]
         <input.file>
         <output.file>
         <rate>
```

The `-c` option instructs the application to compress `input.file` and output the result as the texture `output.file`.

`rate` indicates the block size. You can input this in the following ways:

- A block size such as 5x4 or 3x3x3.
- A *bits per pixel* (bpp) target in the range 8.0 to 0.8 for 2D textures.
- A bpp target in the range 8.0 to 0.6 for 3D textures.

Decompressing a texture

The *astcenc* application provides an option to decompress a texture.

To decompress an ASTC texture use the *astcenc* command with the `-d` option:

```
astcenc -d [options]
         <input.file>
         <output.file>
```

The `-d` option instructs the application to decompress the texture `input.file` and output the result as `output.file`.

Testing texture compression

The *astcenc* application provides an option to test the compression of a texture.

To test the compression of a texture use the *astcenc* command with the `-t` option:

```
astcenc -t [options]<input.file>
         <output.file>
```

The `-t` option instructs the application to compress `input.file` to a texture, then decompress it to the image file `output.file`. You can use this option to test different compression options to see how they work.

Preset compression options

The *astcenc* application includes a number of preset quality options that enable you to trade off quality against compression speed.

astcenc includes the following preset quality options:

- `-veryfast.`
- `-fast.`
- `-medium.`
- `-thorough.`
- `-exhaustive.`

You can use the quality presets to give a high-level hint to the compressor, the individual quality factors are derived from this.



The veryfast option is almost instantaneous, but only produces good results for a small subset of input images.

The exhaustive option takes a long time, but often produces results that have very little visible difference to files compressed in thorough mode.

Compressing normal maps

The *astcenc* application includes options to compress normal maps, or maps used as a data source.

It has traditionally been a problem to compress normal maps, or maps used as a data source rather than color information. *astcenc* includes options for compressing normal maps that implement a number of additional settings and add weight to angular errors that are more important in normal maps.

The options are:

- `-normal_psnr`: Use to compress two channel normal maps.
- `-normal_percep`: Use to compress two channel normal maps. However, this option has slightly different weightings that are designed to provide a better perceptual result.
- `-mask`: This option tells the compressor that the input texture has entirely unrelated content in each channel. Errors in one channel must not affect other channels.



You must only specify one of these options.

Other color formats and texture types

The *astcenc* application supports sRGB, HDR, and 3D textures.

sRGB color space conversion

The *astcenc* application supports non-linear sRGB color space conversion both at compression and decompression time.

To keep images in sRGB color space until the point that they are used, compress them normally then use the sRGB texture formats when you load them. There is an sRGB equivalent of every RGBA format.

As an alternative to using sRGB texture types at runtime, there is a command-line argument that instructs the compressor to transform textures into linear RGBA prior to compression.

The `-srgb` argument converts the color space and compresses the texture into linear space. You can load the texture with the usual RGBA texture formats.

HDR image formats

The *astcenc* application supports HDR image formats.

Using HDR image formats does not require any additional effort in code.

The encoder does not use HDR encoding by default when it is encoding an image in an HDR format. To use an HDR encoding use one of the following arguments:

- `-forcehdr_rgb`: The encoder uses an HDR or LDR as appropriate on a per-block basis. The alpha channel, if present is always encoded as LDR.
- `-forcehdr_rgba`: The encoder uses an HDR or LDR as appropriate on a per-block basis.

There are also simpler versions of these commands that include alterations to the evaluation of block suitability better suited to HDR images:

- `-hdr`
- `-hdra`

The following commands are also similar but check for logarithmic error. Images encoded with these settings typically produce better results from a mathematical perspective but are not as good for perceptual artifacts:

- `-hdr_log`
- `-hdra_log`

3D textures

The *astcenc* application supports 3D textures at a block level.

There are not many widely used 3D image formats to accept as input, so encoding a 3D texture requires a special syntax:

`-array <size>`

This requires a series of 2D files. You provide the input filename and the number of files.

The files must be called `FileN` where `N` is a number incrementing from 0 to `size-1`.

For example, if the input file is `slice.png` with the argument array set to 4, then *astcenc* attempts to load files named `slice0.png`, `slice1.png`, `slice2.png`, and `slice3.png`. The presence of multiple texture layers is taken as an instruction to use a 3D block encoding for the requested rate.

Advanced compression options

The *astcenc* application includes a number of advanced options for controlling image compression.

About advanced compression options

The ASTC compression algorithm works by comparing the block being compressed to algorithmically generated blocks. This can take a long time because there are a very large number of possible blocks.

The *astcenc* application uses a form of guided trial and error to find the best block match. The compression settings define how many blocks are compared and how close to, or far from, a match the block can be.

There are many available options to set various compression quality factors including:

- The number and scope of block partition patterns to attempt.
- Various signal-to-noise cut-off levels.
- The maximum iterations of different bounding color tests.

There are two main categories of argument:

- Quality metrics.
- Search parameters.

Quality settings

To ensure high quality, adjust the quality metrics. That is, alter the factors used by the compressor to judge the quality of a block.

The quality metrics are:

Channel weighting

The simplest quality factors are channel weighting. You can adjust the channel weighting using the command-line argument:

```
-ch <red-weight><green-weight><blue-weight>  
<alpha-weight>
```

The channel weighting defines the weighting values for errors. For example, the following command defines the error on the green channel as four times more sensitive than errors on the other channels:

```
-ch 1 4 1 1
```

The following example is similar but the total error is lower so the result is produced earlier:

```
-ch 0.25 1 0.25 0.25
```



Note

You can combine channel weighting with the swizzling argument `-esw`. For example, for textures without alpha, the swizzle `-esw rgb1` saturates the alpha channel and does not count it in noise computations.

Block weighting

Pixels in blocks can have relatively low errors between them, but if the errors on pixels at the edge of adjacent blocks are in different directions the blocks can be visible.

You can control the errors between adjacent blocks with block weighting. For example: `-b <weight>`



Note

Block weighting can reduce the appearance of blocky images but can increase the overall noise level. Block smoothing between blocks can be very useful for normal maps.

Search parameters

The *astcenc* application uses trial and error to find the best block match. The search parameters define how many blocks are compared, and how good a match the block must be.

The search parameters are:

- `plimit`: The maximum number of partitions tested for each block.
- `dblimit`: This is the maximum Perceptual Signal to Noise Ratio (PSNR) for a block. If a block exceeds this ratio, the algorithm uses it.



Note

This might not be reached if other limits from the settings are hit first.

- `oplimit`: This compares the errors between one and two partitions. If the errors are greater, three or four partition blocks are not tested.

- **mincorrel**: This sets the similarity of colors that the algorithm tries to fit on a single color plane. The lower the value, the wider the range of colors possible in a single partition. The block is not tested with higher numbers of partitions.
- **bmc**: This is the maximum number of block modes to attempt. The block mode defines how the individual color values are precision weighted using different binary modes for each partition.
- **maxiters**: The maximum number of refining iterations to colors and weights for any specific partition attempt.

You can set these values individually to extend searching in specific directions. For example:

- If your texture has a lot of subtle detail, use a high oplimit to ensure subtle color variations do not get missed.
- If your texture has busy randomized patterns, set a higher plimit to search more partitions for a good match.

The following table shows the settings used for the quality presets:

Table 7-5: Preset quality settings

Preset	plimit	dblimit	oplimit	mincorrel	bmc	maxiters
veryfast	2	18.68	1.0	0.5	25	1
fast	4	28.68	1.0	0.5	50	1
medium	25	35.68	1.2	0.75	75	2
thorough	100	42.68	2.5	0.95	95	4
exhaustive	1024	999	1000.0	0.99	100	4

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.

Differences between compute shaders, vertex and fragment shaders

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

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 support up to 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

The following is a simple example of how to implement compute shaders within your application.



The following example is for illustration purposes only. Use a vertex shader in a working project.

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 $2 \times 2 \times 1$ groups, giving 4. However, the real number of threads executed is $4 \times [8 \times 8]$ and this results with a total number of 256 threads. The 8×8 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.



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 geometry shaders.

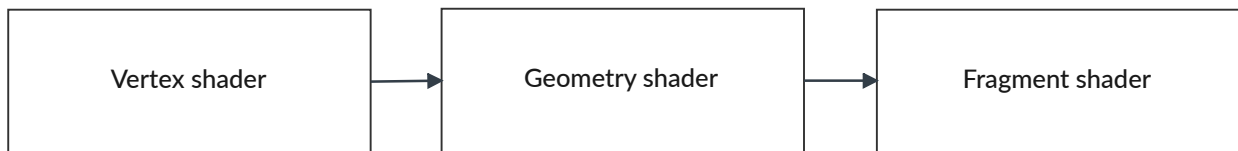
9.1 About geometry shaders

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 the *Android Extension Pack* (AEP). Geometry shaders are typically associated with OpenGL on desktops, but with the AEP you can use them on Android devices with a Mali GPU.

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.

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:

```
layout(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.

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.

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.

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.

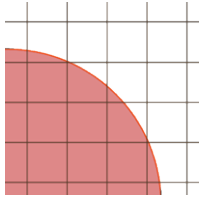
Basic 2D surface net approximation

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.

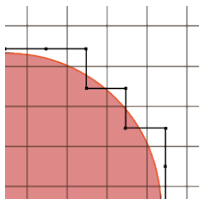
The following figure shows a simple 2D isosurface:

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.

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.

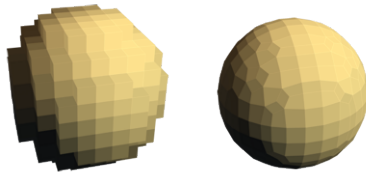
Figure 9-4: A smoothed approximation of a 2D isosurface.

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.



Further Information

[Constrained Elastic SurfaceNets: Generating Smooth Models from Binary Segmented Data.](#)

[Smooth Voxel Terrain \(Part 2\).](#)

[Polygonise \(Paulbourke.net\).](#)

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.

Linking points in 3D

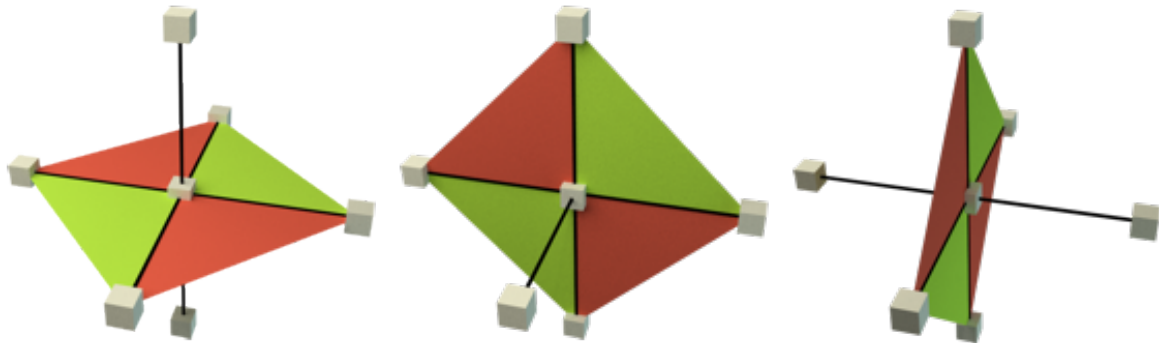
Linking the point on a 2D grid is easy to visualize. However, extending this linking concept to 3D can be harder to visualize.

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.



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.

Flat terrain starting point

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

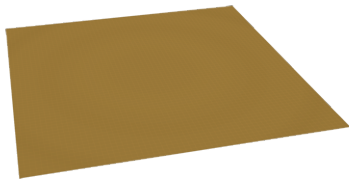
The following code shows how this surface can be described:

```
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.



Oscillating terrain

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

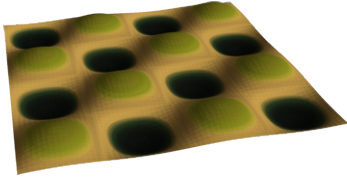
The following code shows how an oscillating surface can be defined:

```
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.



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.

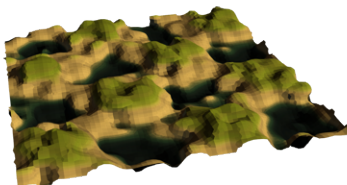
The following code shows an example implementation of noise in a geometry shader:

```
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.



Flattening the tops of the terrain

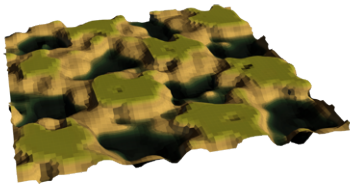
You can alter a geometry shader function so that the tops of the resulting hills in the terrain are flat.

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.



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 computed 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.

Further reading

[Distance functions.](#)

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.

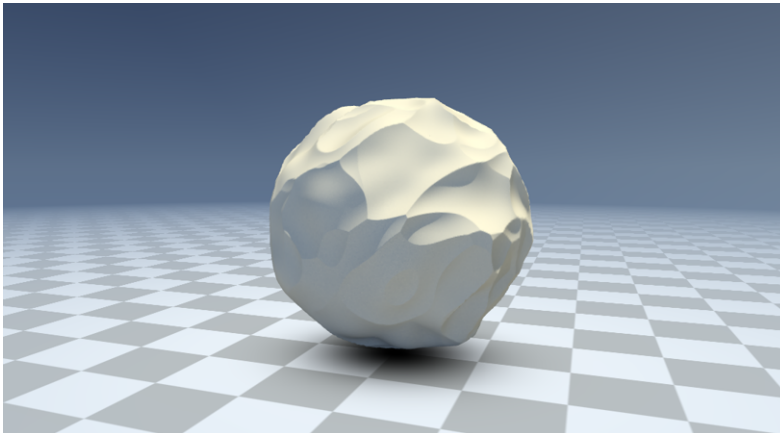
10.1 About tessellation

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:

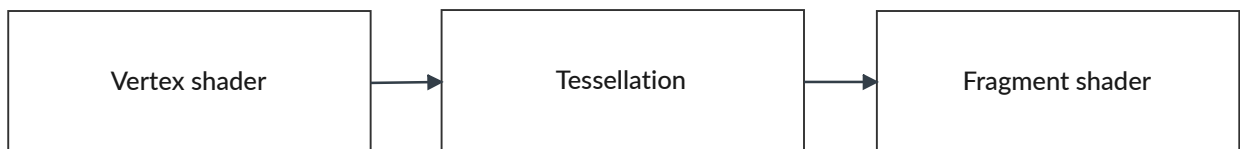
Figure 10-1: An example result of tessellation.



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:

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.

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.

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.

How tessellation works

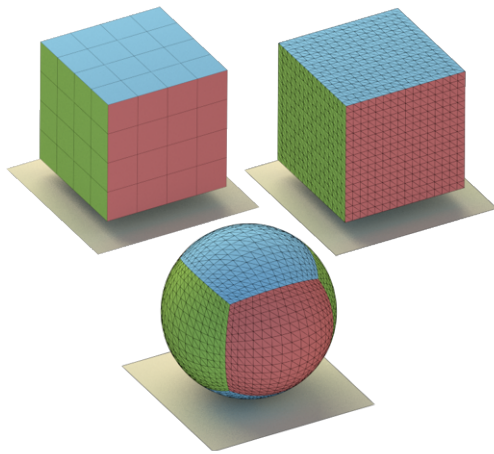
Tessellation splits primitives up into a larger number of smaller primitives. It then moves the new vertices to create more complex geometry than existed originally.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

Combining adaptive tessellation techniques

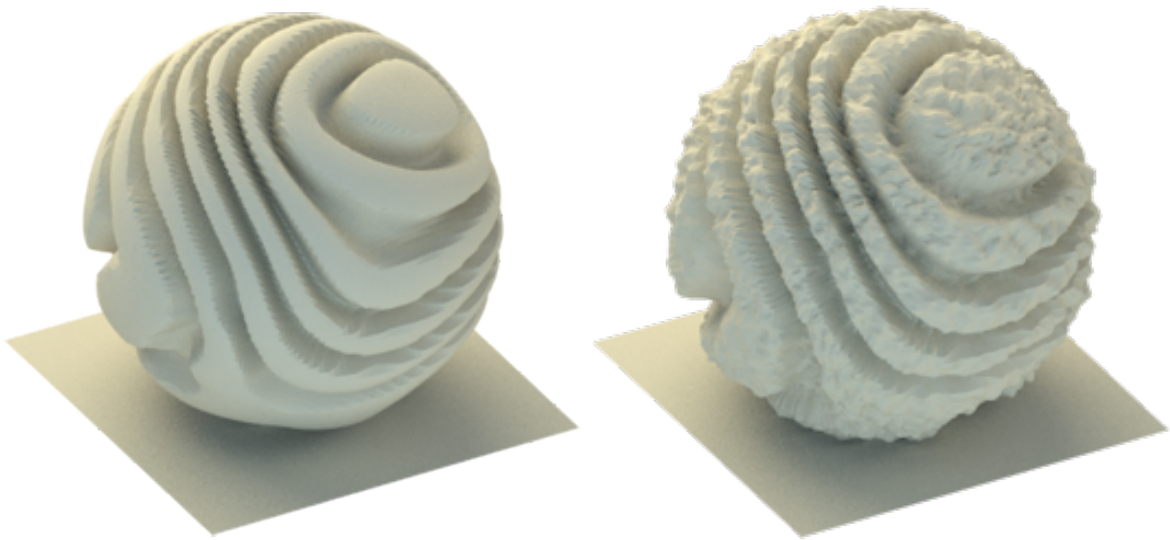
You can combine different adaptive tessellation techniques to maximize the effect. To do this, create a weighted average of the result of the different analysis stages and adjust the tessellation value using it. You can vary the weighting of the average to produce the effect you want.

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 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.`