Accelerating 2D Applications

Version 1.0

Non-Confidential

Issue 01

102524_0100_01_en

Accelerating 2D Applications

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	5 October 2021	Non-Confidential	First release

Proprietary Notice

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.

Product Status

The information in this document is Final, that is for a developed product.

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.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide describes how to improve a device's battery life by reducing device energy use, increasing application performance, and preventing thermal throttling by using specific aspects of 3D rendering to accelerate performance in 2D applications.

This guide was created to help application developers learn how to optimize the performance of their 2D application on a mobile device.

By the end of this guide, you will understand the main causes for inefficiencies in 2D rendering, how to exploit the 3D Depth Testing tool for performance gains, the benefits of combining 2D sprites and 3D geometry in a 2D scene, and how to combine this technique for 2D User Interfaces (UI) in 3D scenes.

Before you begin

Before starting this guide, we recommend that you have a firm understanding of the following concepts:

- Back-to-front render ordering.
- Front-to-back render ordering.

2. Inefficiency in 2D Content

Most of the content displayed on mobile screens today still uses 2D sprite layers, or UI elements to build what is shown on-screen. While most of these applications use OpenGL ES for rendering, only a few applications use the 3D functionality that is found within the GPU.

In 2D games there is not much to optimize. This is because OpenGL ES fragment shader processes are usually trivial, for example interpolate a texture coordinate, load a texture sample, and blend to the framebuffer.

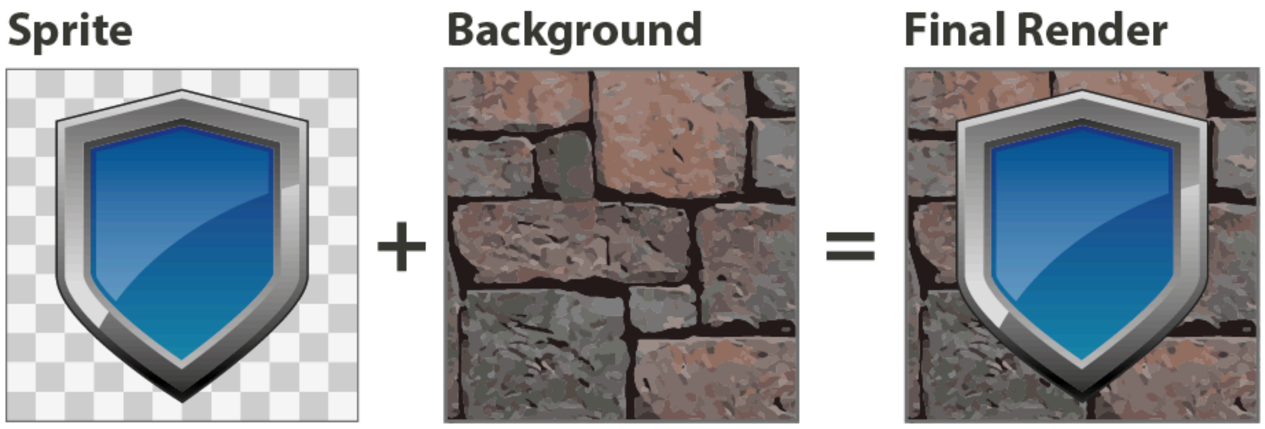
Any performance optimizations for 2D content consist of reducing work redundancy by, for example, removing the need for the shader to run for some of the fragments.

This following example shows a typical blit of a square sprite that is drawn over a background layer.

The outer parts of the shield sprite are transparent. However, the border region is partially transparent, so that it fades cleanly into the background without any aliasing artifacts. Finally, the body of the sprite is opaque.

The following image shows a typical 2D render comprising of a single sprite (with alpha channels) in the foreground, and a 2D texture in the background:

Figure 2-1: Typical 2D render



There are two main sources of inefficiency here:

1. The outer region around this sprite is totally transparent and does not impact the final render at all. However, it still needs time to process.
2. Because the middle part of the sprite is totally opaque, it obscures the background pixels directly beneath it. Because the alpha comes from the texture sample, the graphics driver cannot know ahead of time that the background will be obscured. However, these background fragments are still rendered by the GPU. Therefore, this approach wastes valuable processing cycles rendering for something that is not visible in the final on-screen frame.

This is a relatively basic example with only a single layer of overdraw. However, Arm regularly sees real applications in which over half of all rendered fragments on a 1080p screen are redundant.

If applications use OpenGL ES in a different way to remove this redundancy, then the GPU could render the applications faster, or use the performance headroom created to reduce the clock rate and operating voltage. This saves a substantial amount of energy.

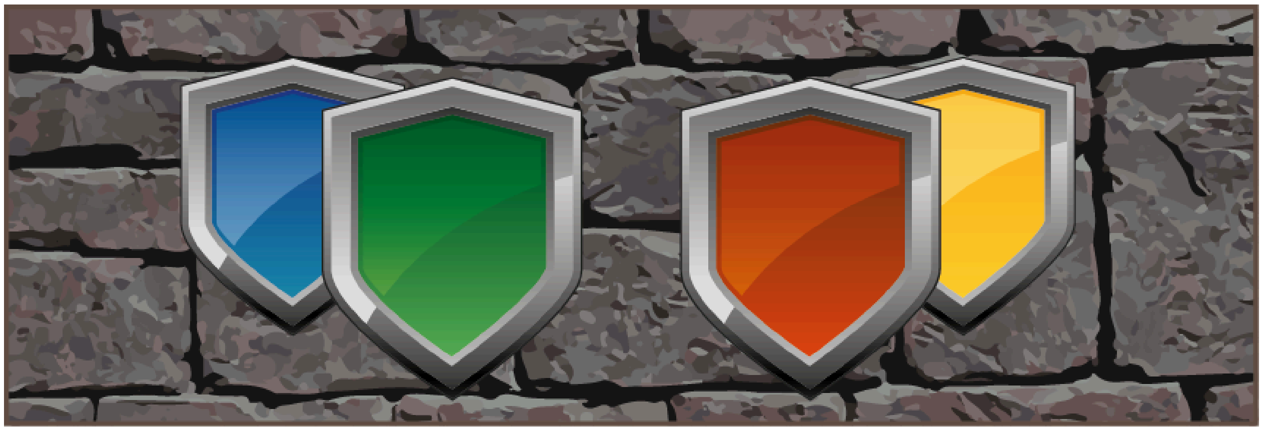
In this guide, we will describe how you can achieve this.

3. Test scene

In our example, we will render a simple test scene consisting of a cover-flow style arrangement of the shield icon. However, this technique works for any sprite set with opaque regions.

The following image shows what our test scene render looks like:

Figure 3-1: Mali Shield Scene



You can see that each shield icon is a square sprite that uses alpha transparencies to hide the pieces that are not visible.

4. Exploiting Depth Testing

In traditional dedicated 2D rendering hardware, the application has to render the sprite layers from back-to-front, to ensure that blending functions correctly.

In our example, the application is using a 3D API to render a 2D scene. What additional tools does the 3D API offer to reduce redundancy? The main tool used in removing redundant work from a 3D scene render is called the depth test.

Every vertex in a triangle has a Z component in its position, which is output from the vertex shader. This Z value encodes how close that vertex is to the camera. The rasterization process interpolates the vertex values, and assigns a depth to each fragment that needs fragment shading.

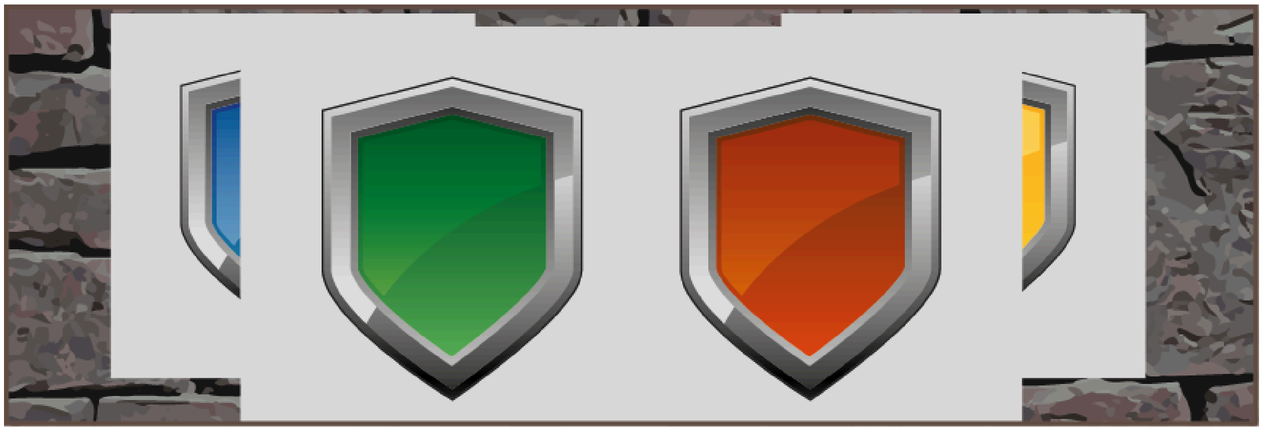
This fragment depth value can then be tested against the existing value stored in the depth buffer. If it is not closer to the camera than the data already in the framebuffer, then the GPU discards the fragment without ever submitting it to the shader core for processing. This is because the GPU now knows that the fragment is not needed.

Sprite rendering engines already track the layering of each sprite, so that they stack correctly for alpha blending. We can now map this layer number to the Z coordinate value assigned to the vertices of each GPU-assigned sprite, and render our scene as if it has 3D depth.

We then use a framebuffer with a depth attachment, enable depth writes, and finally render the sprites and background image in a front-to-back order. The depth test removes the parts of the sprites and the parts of the background which are hidden behind other sprites.

Now, if we run this for our simple test scene, we get the following image:

Figure 4-1: Mali Shield Scene



You can see that the square sprite geometry does not exactly match the shape of the opaque pixels. The transparent parts of the sprites, that are closest to the camera, are not producing any color values because of the alpha test. However, the transparent parts of the sprites are still setting a depth value.

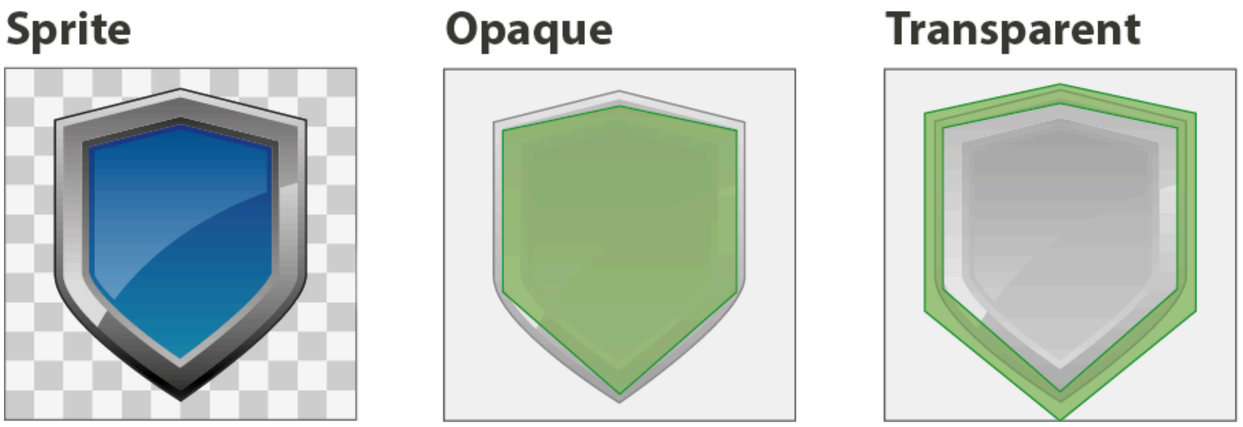
When the sprites on a lower layer are rendered, the depth testing means that the pieces that should be visible underneath the transparent parts of an earlier sprite are being incorrectly removed. As a result, only the OpenGL ES clear color is showing.

5. Sprite Geometry

To fix the sprite backgrounds showing up as opaque, you need to use better sprite geometry. When rendering front-to-back, you can only safely set the depth value for the pixels in the sprite that are totally opaque. Therefore, the sprite atlas needs to needs to be used with two different sets of geometry for each sprite.

The following images show this. The first set of geometry, which is indicated by the green area in the middle image below, covers only the opaque geometry. The second set of geometry, indicated by the green area in the right-hand image below, picks up everything else that is not totally transparent. Any area that is fully transparent can be dropped completely.

Figure 5-1: Mali Shield Scene



Vertices are computationally expensive, so use as few vertices as possible when generating these geometry sets. And while the opaque region can only contain totally opaque pixels, the transparent region can safely contain both opaque pixels and totally transparent pixels without any side-effects. This means that you can use rough approximations for a fit that you feel is good enough.

Remember that, for some sprites, it is not worth generating the opaque region at all. This is because there may be no opaque texels, or the area involved may be too small. So, some sprites may consist of only a single region rendered as a transparent render.

In most cases, if your opaque region is smaller than 1024 pixels, it is probably not worth making the geometry more complex. However, it is worth experimenting for your own use case.

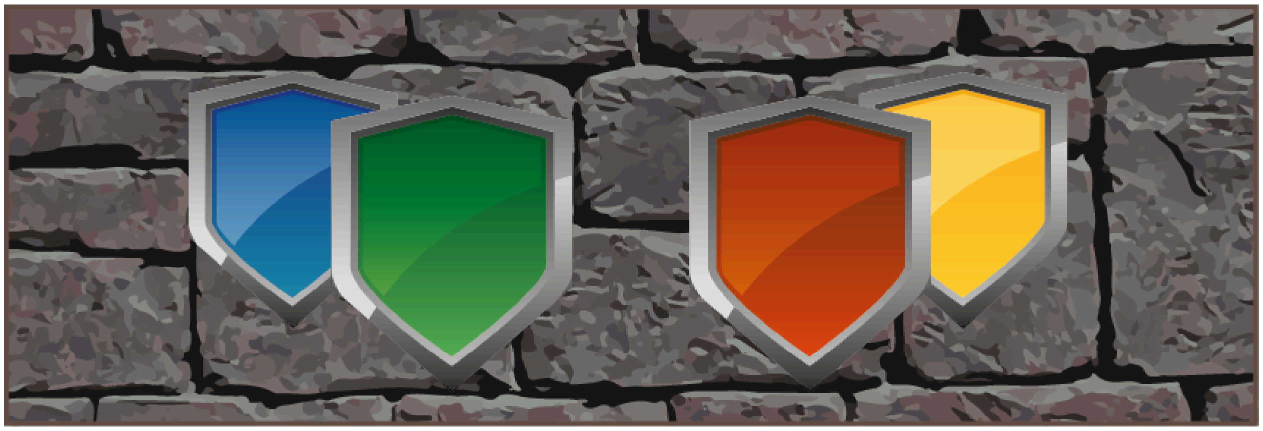
Generating additional geometry can be difficult, but sprite texture atlases are normally static. Therefore, this can be done offline as part of the application content authoring process, and does not need to be done live on the platform at runtime.

6. Draw Algorithm

In the previous section we described how to create the two geometry sets for each sprite. Now we can render the optimized version of our test scene. First, we render all the opaque sprites regions, then we render the background - using front-to-back rendering - with both depth testing and depth writes enabled.

This results in the output shown in the following image:

Figure 6-1: Mali Shield Scene

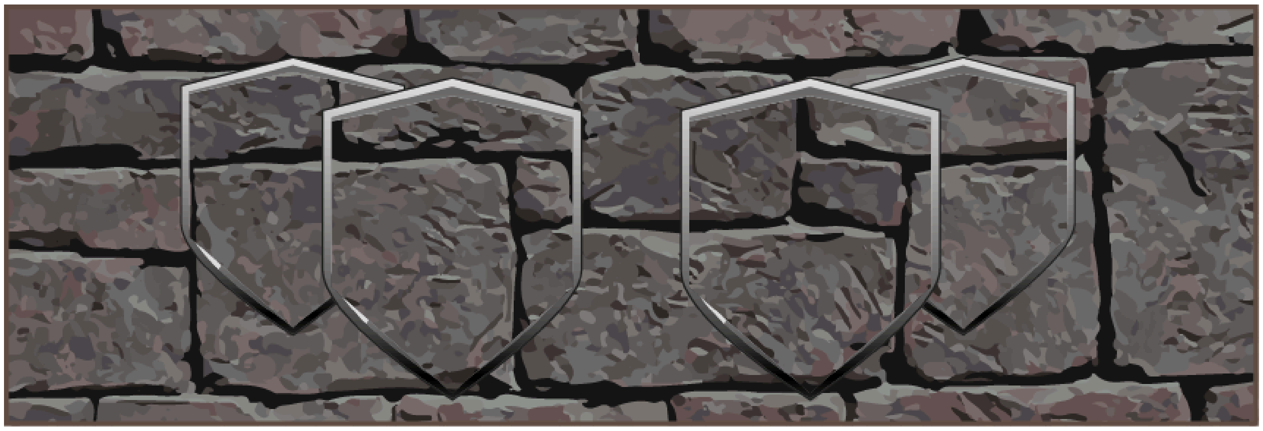


The renderer has not calculated or drawn the pixels for each sprite, or the part of the background, that is hidden underneath another opaque sprite. This is because the early depth test was performed before the shading calculations occurred.

With the opaque geometry rendering complete, it is now possible to render the transparent region for each sprite in a back-to-front order. Depth testing must be left on, so that sprites on a lower layer do not overwrite an opaque region from a sprite that has already been rendered in a higher layer. If you want to save a little bit of power, you can disable depth buffer writes.

To show the additional rendering added by the pass, clear the color output of the opaque stage, keep its depth values, and then draw the transparent pass. The result of this can be seen in the following image:

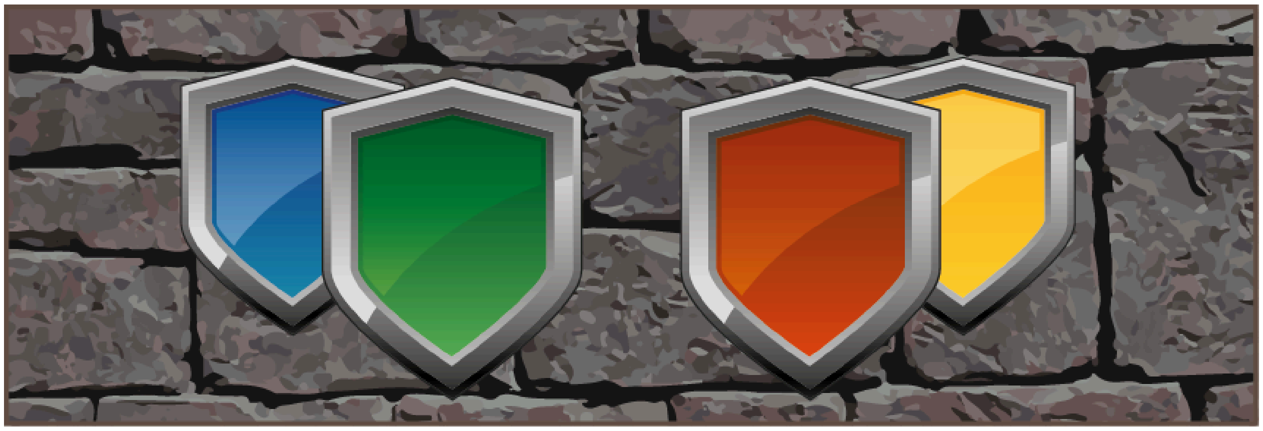
Figure 6-2: Mali Shield Scene



Areas in which one of the outer rings is obscured indicates where rendering work has been saved. The depth test has removed the missing parts.

If we put the scene all back together, and render both passes to the same image, then we arrive back at the same visual output as the original back-to-front render, which you can see again in the following image:

Figure 6-3: Mali Shield Scene



The new render results in around 35% fewer fragment threads started, equaling around a 35% drop in MHz required to render this scene.

The final bit of operational logic that we needed is to ensure that the depth buffer that was added to the scene is not written back to memory. If your application is rendering directly to the EGL window surface, then there is nothing to do here. This is because depth is implicitly discarded for window surfaces automatically.

However, if your engine is rendering to an off-screen Draw , then you must add either of the following OpenGL ES calls before changing the FBO binding away from the off-screen target:

- OpenGL ES 3.0 or newer: `glInvalidateFramebuffer()`

- OpenGL ES 2.0 only: `glDiscardFramebufferEXT()`

7. Use in 3D Scenes

Depth testing can also be used when rendering the 2D User Interface (UI) elements in 3D games. Render the opaque parts of the UI with a depth that is very close to near the clip plane, before rendering the 3D scene.

Now render the 3D scene as normal, and any parts behind the opaque UI elements will be skipped. Finally, the remaining transparent parts of the UI can be rendered and blended on top of the 3D output.

To ensure that the 3D geometry does not intersect the UI elements, `glDepthRange()` should be used to limit the range of depth values emitted by the 3D pass very slightly. This approach guarantees that the UI elements are always closer to the near clip plane than the 3D rendering.

8. Next steps

This guide has looked at how the use of depth testing and depth-aware sprite techniques can be used to significantly accelerate 2D scene rendering using 3D graphics hardware.

Remember that adding vertices to provide the partition between opaque and transparent regions of each sprite does create additional computational complexity. Therefore, be careful to minimize the number of vertices used for each sprite. Otherwise, the costs of the extra vertex processing and small triangle sizes will outweigh the benefits.

For cases in which the additional geometry required to fit the sprint is too complicated, or the screen region covered is too small, simply fall back to back-to-front rendering on a per-sprite basis by omitting the opaque geometry and rendering the whole sprite as transparent.