Article

Tailor your apps for Apple GPUs and tile-based deferred rendering

Learn about characteristic Apple GPU features, including imageblocks, tile shaders, and raster order groups.

Overview

The GPUs in Apple silicon implement a rendering technique called *tile-based deferred rendering* (TBDR) that optimizes performance and power efficiency. The GPU breaks up the render destination into a grid of smaller regions, called *tiles*. It processes each tile with one of its GPU cores, often running many at the same time. The GPU defers, or postpones, the rendering phase for each tile until after it evaluates all the geometry for that tile.

Starting with A11, Apple-designed GPUs deliver several features that significantly enhance TBDR. Your app can access these enhancements through the Metal API, helping your apps and games achieve new levels of performance and capability. The features include imageblocks, tile shading, raster order groups, and imageblock sample coverage control. A11 and later Apple GPUs also improve fragment discard performance and simplify implementing techniques, including subsurface scattering, order-independent transparency, and tile-based lighting algorithms.

Draw more with tile-based deferred rendering

Your app can draw more complicated scenes on a TBDR GPU because it significantly improves performance over traditional, immediate-mode (IM) GPUs by saving time, energy, and memory bandwidth. For example, an IM GPU fully processes primitives, such as lines and triangles, regardless of whether or not they're visible in the rendering.

A TBDR GPU avoids doing unnecessary work by processing all of the geometry of a render pass at the same time and shading only the visible primitives. The GPU splits the work into tiles so it can separately and simultaneously process all the geometry that intersects each tile and discard any

occluded (or concealed) primitives. For each tile, the GPU then generates fragments (or potential pixels) from the remaining, visible primitives, processes them with a fragment shader, and writes them into *tile memory*. Tile memory is fast, temporary storage that resides on the GPU itself. After the GPU finishes rendering each tile into tile memory, it writes the final result to device memory.

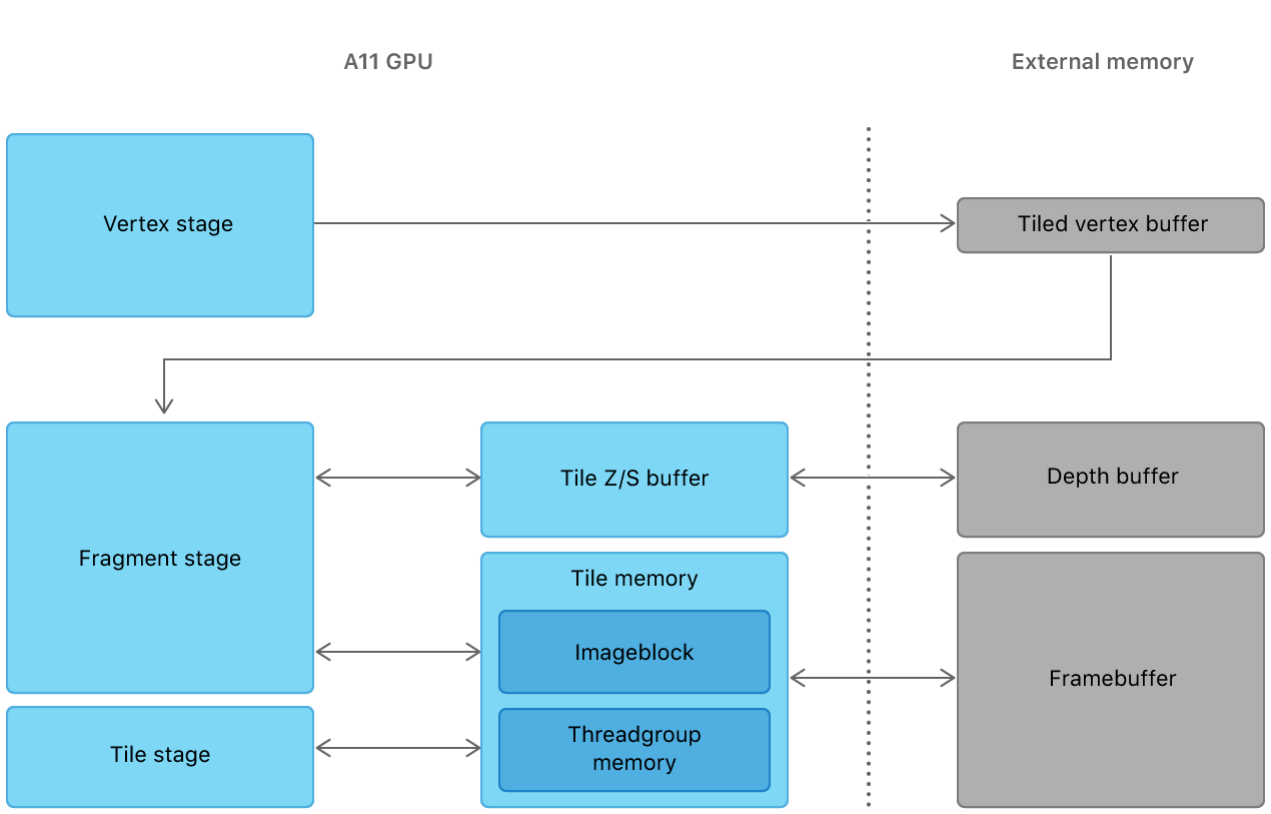
Tile memory is an important component to TBDR because it saves time and energy by avoiding accessing device memory as much as possible. A fragment shader core's access to tile memory has important advantages over the GPU's access to device memory, including the following:

- Bandwidth that's many times faster than device memory
- Access latency that's many times lower than device memory
- Energy consumption that's significantly less than accessing device memory

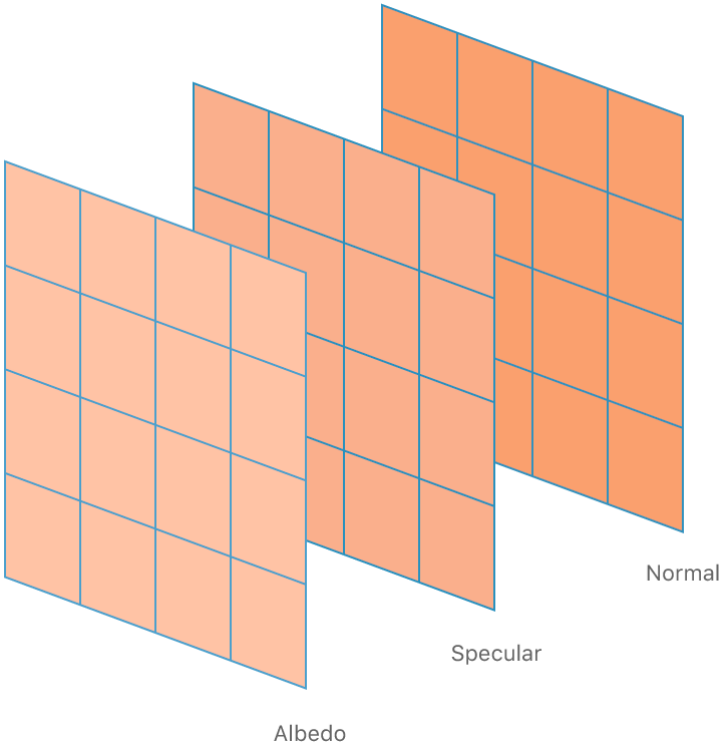
While the GPU runs the final stages of a render pass to tile memory, it can start the vertex stage of a future render pass. The GPU can use more hardware blocks at the same time by running both stages in parallel because they tend to use different compute and memory components.

Add customizations to your fragment shaders with imageblocks

You can define and manipulate custom per-pixel data structures in *imageblock memory*, a high-bandwidth memory region in the GPU. Imageblocks are tiles of structured image data stored in local memory, allowing you to describe image data in tile memory that Apple GPUs can manipulate efficiently. They're deeply integrated with fragment processing and the tile shading stage, and are also available to compute kernels. Metal has always rendered to imageblocks on devices with Apple silicon, but starting with the A11 GPU, Metal gives you complete control over data structures in imageblocks. An imageblock can pass data between the fragment and tile stages of a render pass. Threadgroup memory is suitable for unstructured data, but an imageblock is more suitable for image data.



An imageblock is a 2D data structure with a width, height, and pixel depth. Each pixel in an imageblock can consist of multiple components, and you can address each component as its own image slice. For example, you can have three image slices that represent albedo, specular, and normal components.



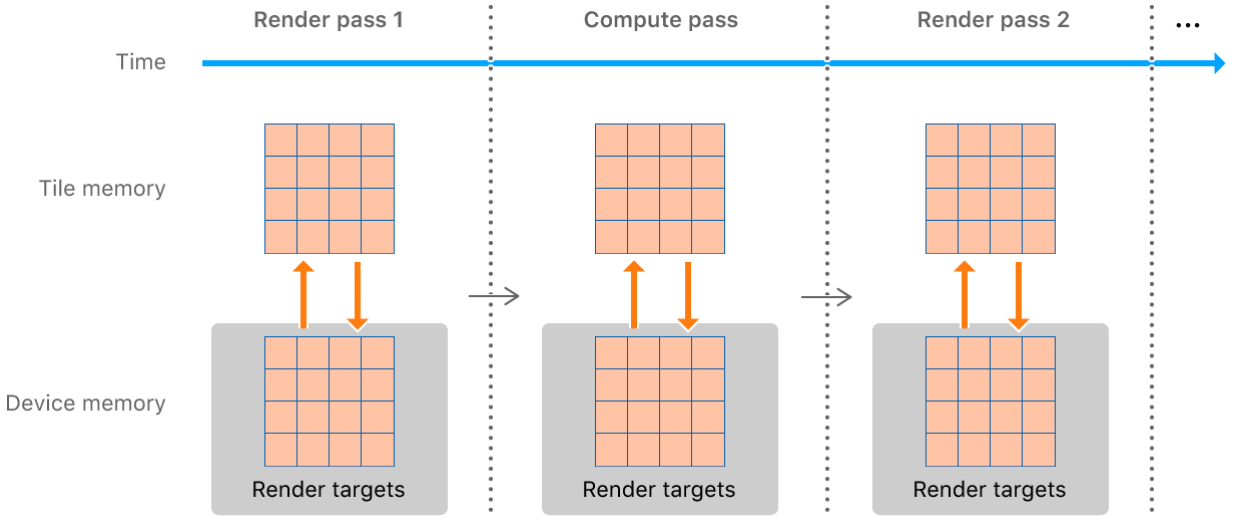
Imageblocks are available to both kernel and fragment functions and persist for the lifetime of a tile, across draws and dispatches. Imageblock persistence means that you can mix render and compute operations in a single rendering pass with tile shaders, where both can access the same local memory. You can create sophisticated algorithms that remain in local GPU memory by keeping multiple operations within a tile.

Your existing code automatically creates imageblocks that match your render attachment formats. However, you can also define your own imageblocks completely within your shader. Imageblocks that you define can be far more sophisticated than those render attachments create. For example, an imageblock can include additional channels, arrays, and nested structures. Furthermore, you can reuse the imageblocks you define for different purposes across different phases of your computation.

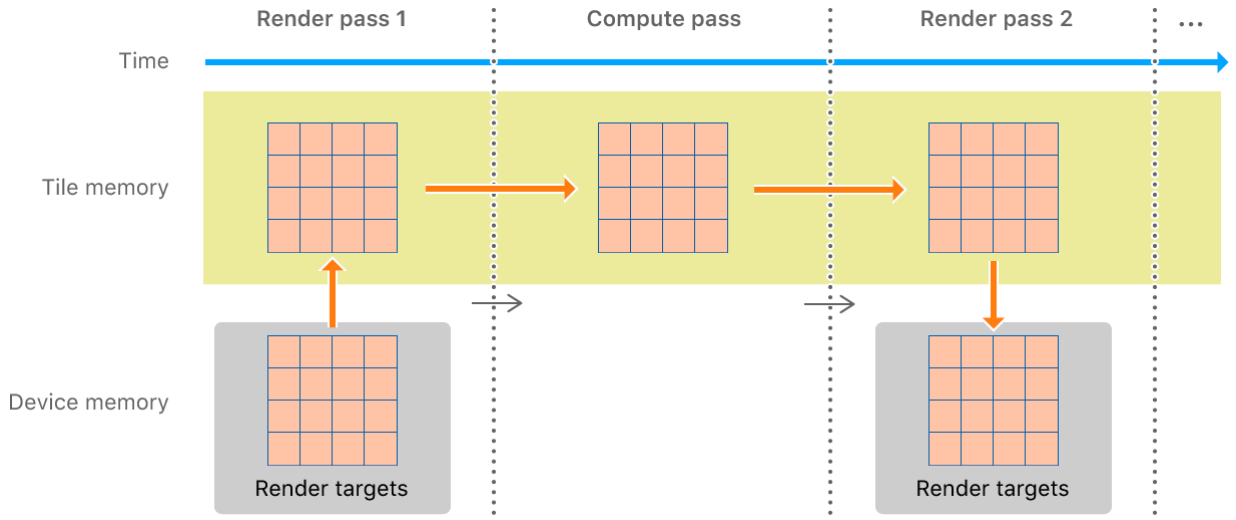
Within a fragment shader, the current fragment has access to only the imageblock data associated with that fragment's position in the tile. In a compute function, a thread can access all of the imageblock data. When rendering with attachments, load and store actions still read and write data in tile memory. However, if you're using explicit imageblocks, use a compute function to explicitly read and write to device memory. The GPU may automatically flush the contents of tile memory to system memory in an efficient block transfer that takes advantage of memory hardware.

Save memory bandwidth across render and compute passes with tile shaders

Tile shading lets you combine rendering and compute operations into a single render pass while sharing local memory. Many rendering techniques require a mixture of drawing and compute commands. Traditional GPUs separate rendering and compute commands into distinct passes. These passes typically can't communicate directly with each other. Apps work around this limitation by saving the results from one pass into device memory and then loading that data back for the next pass. In some scenarios, such as in a multiphase rendering algorithm, apps may copy intermediate data to device memory many times.



Tile shaders are compute or fragment functions that execute as part of a render pass. They allow your app to compute and save data to tile memory that's persistent on the GPU between render passes.

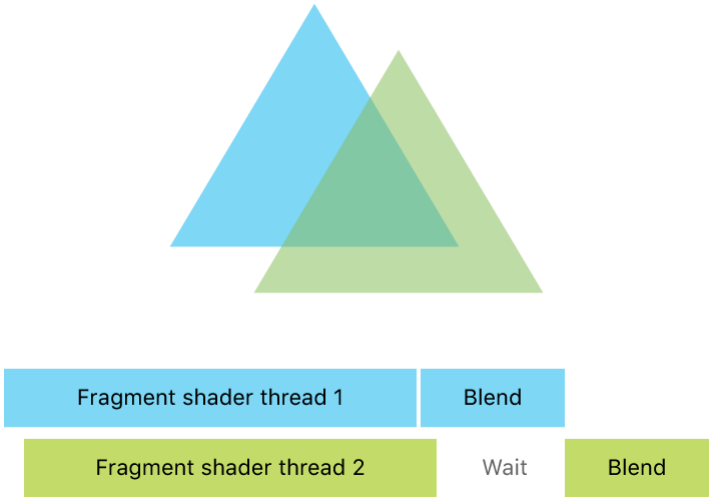


Apps that use tiles shaders can avoid storing intermediate results out to device memory and save time by storing data in faster tile memory.

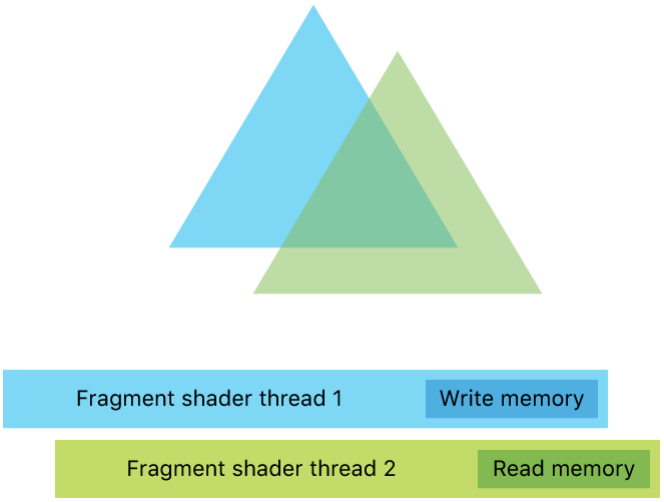
Sequence operations with raster order groups

Your apps can precisely control the order of parallel fragment shader threads that access the same pixel coordinates with *raster order groups*. Raster order groups offer ordered memory access from fragment shaders and simplify rendering techniques, such as order-independent transparency, dual-layer geometry buffers, and voxelization.

Metal guarantees the GPU blends in draw call order, which gives the illusion that the GPU renders the scene sequentially. For example, here's a scene that contains two overlapping triangles. The blue triangle partially obscures the green triangle behind it.

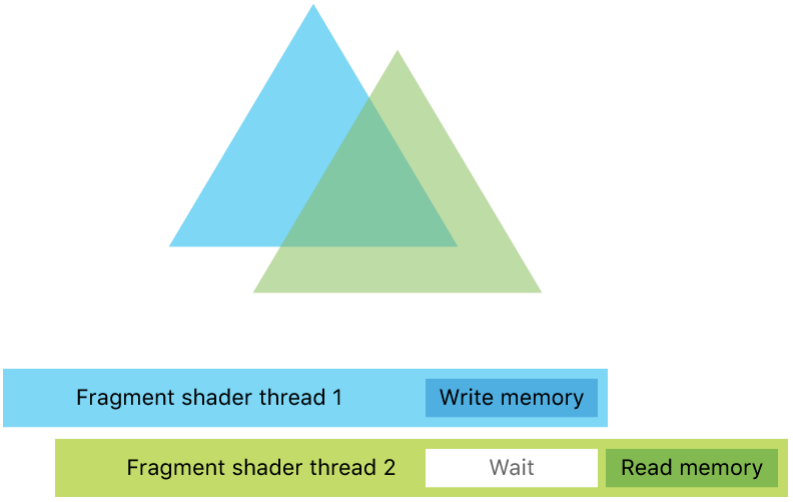


The fragment shaders for each triangle run concurrently on their own thread. The fragment shader for the rear triangle might not execute before the fragment shader for the front triangle, which can be a problem for a shader that needs the results from another triangle's shader for its custom blending function. Because of concurrency, this read-modify-write sequence can create a race condition.



Raster order groups overcome this access conflict by synchronizing threads that target the same pixel coordinates and sample (if you activate per-sample shading).

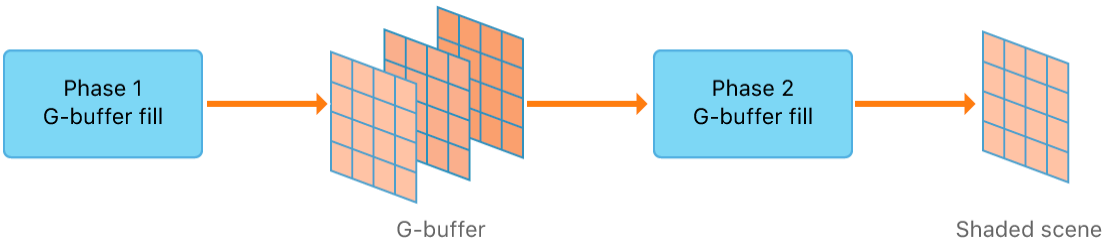
To implement raster order groups, annotate pointers to memory with an attribute qualifier. Shaders that access pixels through those pointers go in per-pixel submission order. The hardware waits for any older fragment shader threads that overlap the current thread to finish before the current thread proceeds.



Metal on recent Apple GPUs extends raster order groups with additional capabilities. They allow you to synchronize individual channels of an imageblock and threadgroup memory. You can also create multiple order groups, which give you finer-grained synchronization and minimize how often your threads wait for access. For example, these raster order groups can improve the performance of the *deferred shading* technique — a popular lighting technique that's not related to TBDR. Traditionally, deferred shading requires two phases:

1. Fill a geometry buffer (or *g-buffer*) and produce multiple textures.

2. Render light volumes by calculating shading results with those textures.



Deferred sharing is memory bandwidth intensive because the shaders write those textures to device memory in the first phase and then read them back in the second phase. You can eliminate the need for the intermediate textures by using multiple order groups to coalesce both render phases into one. To do that, keep the geometry buffer in tile-sized chunks so they can remain in local imageblock memory.

On a traditional GPU, a thread responsible for a secondary light must wait for access until prior threads complete before it can begin. This wait forces the threads to run serially, even if the accesses don't conflict with each other.

You can use multiple order groups to run the nonconflicting reads concurrently by:

1. Adding the three geometry buffer fields — albedo, normal, and depth — to the first group
2. Adding the accumulated lighting result to the second group

Apple GPUs can order the two groups separately so that outstanding writes into the second group don't impede the reads from the first group.

The two threads synchronize at the end of execution to accumulate the lights.

Customize pixel blending with enhanced multisample antialiasing

You can create custom multisample antialiasing (MSAA) algorithms by accessing multisample tracking data within a tile shader. MSAA is a technique that improves the appearance of primitive edges by using multiple depth and color samples for each pixel.

Each pixel has a single sample position that a triangle either covers or doesn't cover, which can make jagged edges at some angles.

However, 4x MSAA smooths the appearance of the jagged edges by sampling each pixel at four different positions. The GPU averages the colors of each sample within the pixel to determine its final color.

The A-Series GPUs have an efficient MSAA implementation. The hardware tracks whether each pixel contains a primitive's edge so that it runs the per-sample blending only when necessary. If another primitive covers all the samples within in a pixel, the GPU blends only once for the entire pixel.

An Apple GPU tracks the number of unique samples (or colors) for each pixel and updates that data as it renders new primitives. For example, consider a pixel that contains the edges of two overlapping triangles, and the sample positions represent three unique colors.

Apple GPUs prior to A11 blend each of the pixel's three covered samples. Starting with A11, Apple GPUs blend only twice because two samples share the same color. In this example, the color at index 1 is a blend of green and pink, and the color at index 2 is pink.

Apple GPUs can reduce the number of unique colors in a pixel. For example, if the GPU renders an opaque triangle on top of the earlier triangles, it represents the pixel by a single color.

In this case, the new triangle covers all of the samples, and the GPU can represent a single color by merging the three colors into one.

You can implement a custom resolve algorithm by modifying the sample coverage data in the tile shaders. For example, consider a complex scene that contains separate render phases for opaque and translucent geometry. You can add a tile shader that resolves the sample data for the opaque geometry before blending the translucent geometry.

The tile shader works on data in local memory and can be part of the opaque geometry phase.

See Also

Apple silicon

- Porting your Metal code to Apple silicon

  Create a version of your Metal app that runs on both Apple silicon and Intel-based Mac computers.